

Prima Esercitazione

7 marzo 2022

Linux shell e
linguaggio C

comandi di uso comune per la gestione del file system

**cd, rm, cp, cat, mv, mkdir,
rmdir, chmod, chgrp, chown**

il comando cd: change directory

È possibile ‘**spostarsi**’ da un direttorio attraverso il comando **cd**. La sintassi è:

cd [<nuovo direttorio>]

- il direttorio destinazione si può esprimere con il nome **relativo** oppure **assoluto**
- se l’argomento non viene specificato, il nuovo direttorio è la **home directory** dell’utente
- per spostarsi all’interno di un determinato direttorio bisogna avere per tale direttorio i **diritti di esecuzione**

il comando cd

Esempio:

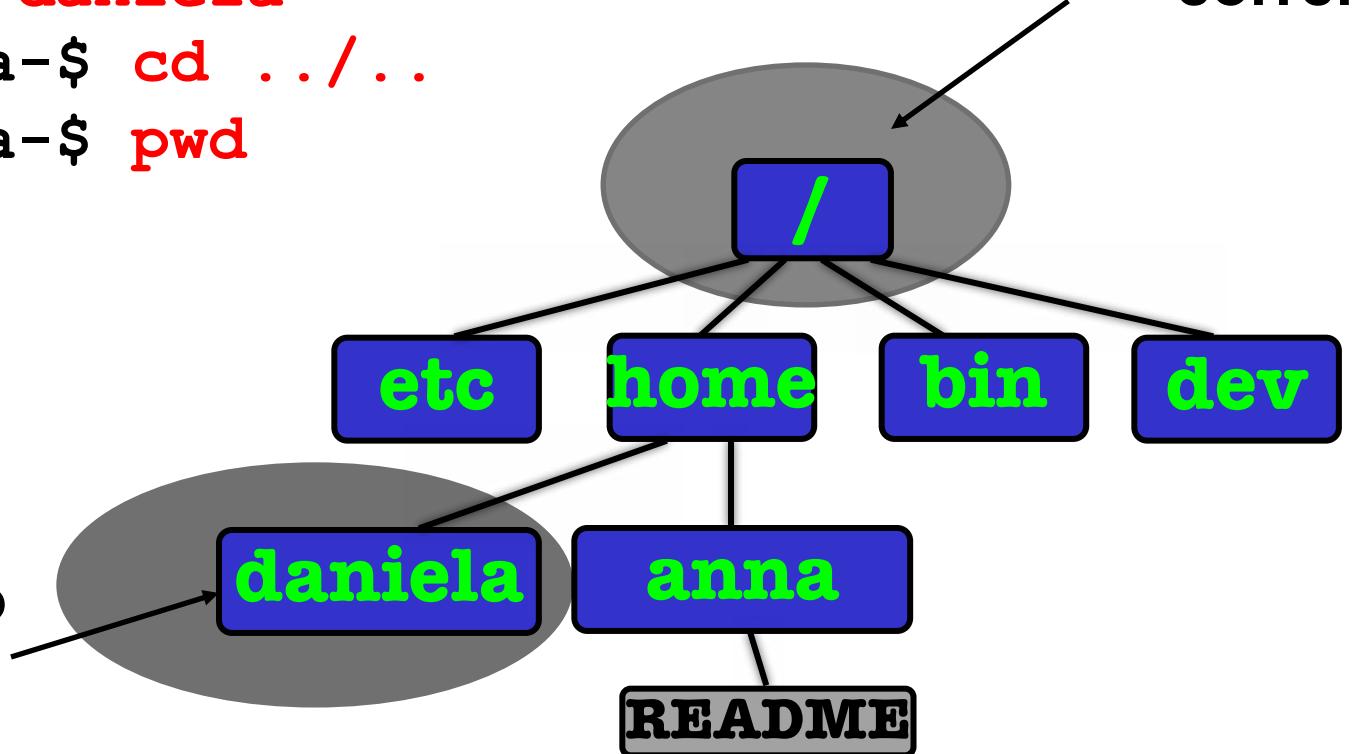
```
daniela-$ pwd  
/home/daniela
```

```
daniela-$ cd ../../..
```

```
daniela-$ pwd  
/
```

direttorio iniziale

nuovo directory corrente



il comando cd

Esempio:

```
daniela-$ pwd
```

/dev

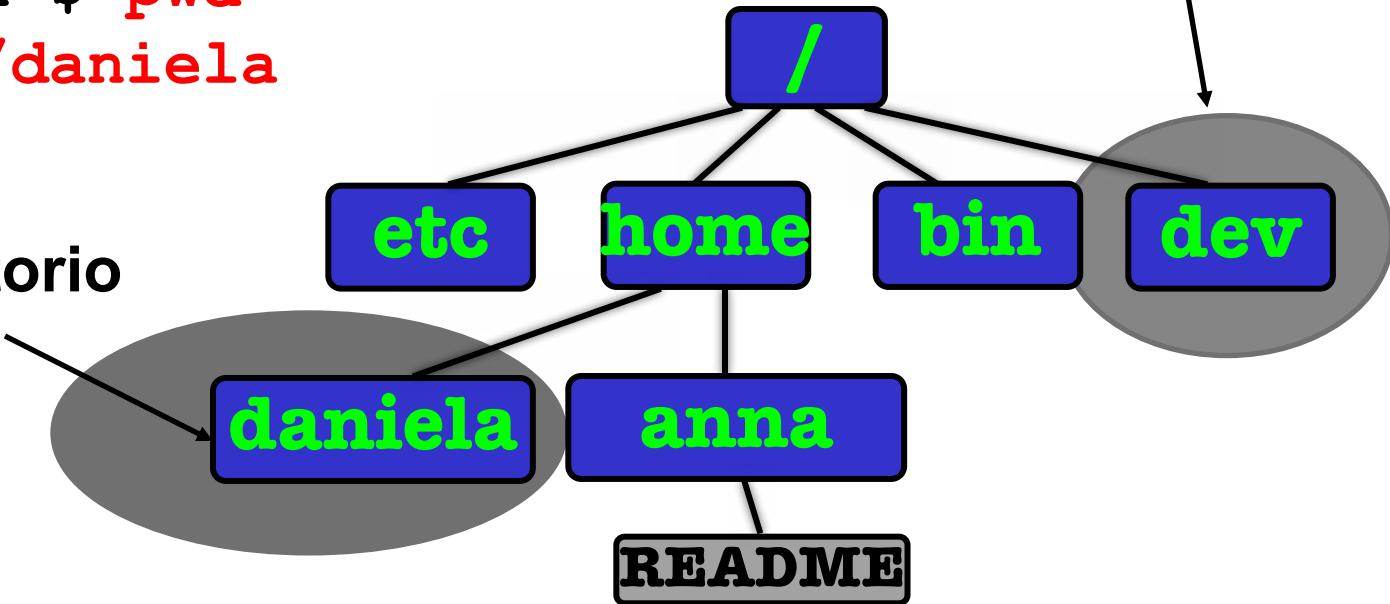
```
daniela-$ cd
```

```
daniela-$ pwd
```

/home/daniela

nuovo directory
corrente

direttorio
iniziale

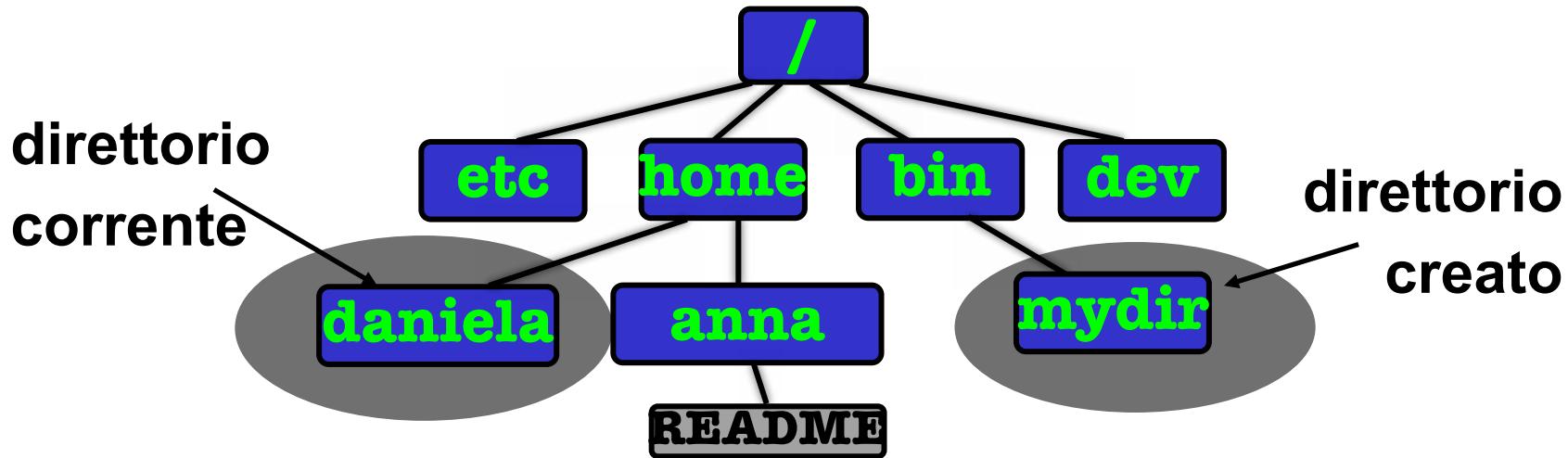


modifica del file system: directory

- **Creazione** di un direttorio: `mkdir <nomedir>`
- **Eliminazione** : `rmdir <nomedir>`
- per creare un direttorio è necessario avere i diritti di **scrittura** nel direttorio all'interno del quale lo si vuole inserire
- per **eliminare** un direttorio è necessario:
 - avere i diritti di **scrittura** sul direttorio di appartenenza del direttorio da eliminare;
 - che la directory sia **vuota**

Esempio di mkdir/rmdir

creo un nuovo directory utilizzando il percorso relativo o assoluto:



```
:~$ mkdir /bin/mydir      occorre avere w su bin  
:~$ mkdir ../../bin/mydir
```

```
:~$ rmdir /bin/mydir      occorre avere w su bin
```

lettura di file di testo

- è necessario avere i diritti di lettura per visualizzare il contenuto di un file di testo
- **cat [<nomefile>...]**: visualizza l'intero file
 - provare: cat file.txt
- **more [<nomefile>...]**: visualizza per videate
- altri comandi:
 - **grep <stringa> [<nomefile>...]**
(ricerca di una stringa in un file),
 - **wc [-lwc] [<nomefile>...]**
(conteggio di righe / parole / caratteri)

cancellazione, copia e spostamento di file

- **copia** di un file (e diritti):

`cp <nomefile> <nuovofile>`

- **spostamento** di un file (e diritti):

`mv <nomefile> <nuovofile>`

- **eliminazione** di un file:

`rm <nomefile>`



esempi

```
:~$ cp file.txt sera (copia il file)
```

```
:~$ ls  
file.txt sera
```

```
:~$ mv file.txt poesia (sposta/rinomina il file)
```

```
:~$ ls  
poesia sera
```

```
:~$ rm poesia  
:~$ ls  
sera
```

NB: Per eliminare un file occorre avere **diritto di scrittura** sulla directory che lo contiene

protezione

proprietà, accessi, bit di
protezione

proprietà di file

- Come abbiamo visto, a ciascun utente viene assegnato uno **username** e una password, e gli utenti sono classificati in **gruppi**. Es:

Username: anna [User-id: 1530]

Group: staff [Group-id: 22]

- ad ogni file è associato lo username ed il gruppo dell'utente **proprietario** (inizialmente, chi lo crea)

```
anna@lab3-linux:~$ ls -l
-rw-r--r-- 1 anna staff 57 Apr 12 13:06 f1.txt
```

The diagram illustrates the output of the `ls -l` command. It highlights the first character of the permissions ('-') and the owner information ('1 anna'). It also highlights the group information ('staff'). Two arrows point from these highlighted areas to two boxes: 'utente proprietario' and 'gruppo proprietario'.

- cambiare la proprietà di un file (assegnandola a un altro utente / gruppo): **chown, chgrp**

accesso ai file

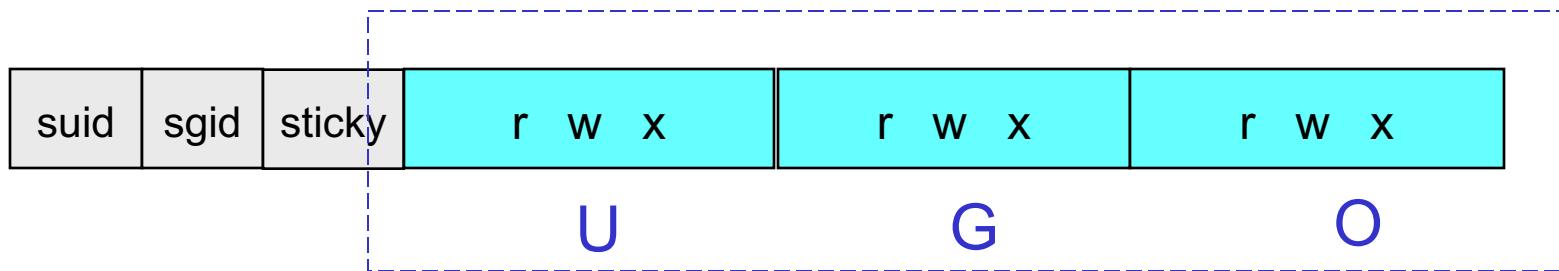
- esistono tre modalità di accesso ai file: lettura, scrittura, esecuzione
- il proprietario può concedere o negare agli altri utenti il permesso di accedere ai propri file
- esiste un utente privilegiato (**root**) che ha accesso incondizionato ad ogni file del sistema

bit di protezione

- Ad ogni file sono associati **12 bit** di protezione:

suid	sgid	sticky	r w x	r w x	r w x
			U	G	O

Bit di Protezione: lettura, scrittura, esecuzione



9 bit di lettura (read), scrittura (write), esecuzione(execute) per:

- utente proprietario (**U**ser)
- utenti del gruppo (**G**roup)
- tutti gli altri utenti (**O**thers)

bit di protezione: lettura, scrittura, esecuzione

Ad esempio, il file:

	U	G	O
pippo.txt	1 1 1	0 0 1	0 0 0
	r w x	- - x	- - -

- è leggibile, scrivibile, eseguibile per l'utente proprietario
 - è solo eseguibile per gli utenti appartenenti al gruppo proprietario
 - nessun tipo di modalità per gli altri
-
- formato ottale: 111 => 7; 001 => 1; ... -rwx--x--- => 0710

modifica dei bit di protezione

- **chown / chgrp** permettono di modificare la **proprietà** di un file (occorre essere root)
- è possibile **cambiare i permessi** dei propri file attraverso il comando chmod:

chmod <mode> <nomefile>

[ugoa] [[+ - =] [rwxXstugoa...] ...] [, ...]

oppure: formato ottale dei bit di protezione



esempio chmod

```
:~$ ls -l sera
-rw-rw-r-- 1 daniela staff ... sera

:~$ chmod 0666 sera ([6]8 = [110]2)
:~$ ls -l sera
-rw-rw-rw- 1 daniela staff ... sera

:~$ chmod a-w,u=rw sera
:~$ ls -l sera
-rw-r--r-- 1 daniela staff ... sera
```



esempi modifica file (diritti)

```
$ ls -l sera  
-rw-r--r-- 1      loreti staff ... sera  
$ chmod 0400 sera          ([4]8 = [100]2)  
$ mv sera subito  
$ ls -l subito      (mv sposta il file e i suoi diritti)  
-r----- 1      loreti staff ... subito  
$ rm subito  
rm: remove 'subito' ,overriding mode 0400[y/n]?
```

Questa domanda è una feature di **rm**, **NON** un fatto di permessi!

Il permesso di cancellare un file dipende dai permessi sulla directory che lo contiene

cancellazione, copia e spostamento di file: diritti

- **rm <nomefile>**
 - Per eliminare un file occorre avere diritto di scrittura sulla directory che lo contiene
- **cp <nomefile> <nuovofile>**
 - Per copiare un file occorre avere diritto di lettura sul file e di scrittura sulla directory di destinazione
- **mv <nomefile> <nuovofile>**
 - Per spostare un file occorre avere diritto di lettura sul file, di scrittura sulla directory di provenienza e su quella di destinazione

Programmare in linguaggio C

editor, compilatore, parametri

Editor

- Esistono vari editor offerti dal sistema:
 - vi: editor standard UNIX (1976)
 - Vim: (vi improved) versione estesa
presente in tutte le distribuzioni unix
 - **Kate**(Applications->Accessories->Kate),
Kwrite
 - Gedit
 - Emacs
 - ...

Compilazione sorgenti - C

- un file.c non è direttamente eseguibile dal processore. Occorre tradurlo in linguaggio macchina compilandolo.
- Comando **gcc <file>**:
 - compila **<file>** producendo il file eseguibile **a.out**
 - per dare nome diverso al file prodotto: opzione **-o**
- Esempio: **gcc file.c -o fex**
- per rendere fex eseguibile: **chmod u+x fex**
- Esecuzione: **./fex <parametri>**

Compilazione sorgenti - C

file.c (file sorgente C)

```
main(int argc, char *argv[])
{ ... }
```

```
:~$ gcc file.c -o fex
```

fex (file in linguaggio macchina)

```
@<<Q?tdR?td` ` ??/lib64/ld-linux-x86-
64.so.2GNUGNU?>;D??c\?S?\%D???Dgp....
```

```
:~$ ls -l fex
```

```
-rw-r--r-- 1 dloreti staff ... fex
```



per rendere fex eseguibile: **chmod u+x fex**

Esecuzione: **./fex <parametri>**



Parametri della linea di comando: *argc, argv*

```
main (int argc, char *argv[]){ }
```

- **int argc:** rappresenta il numero degli argomenti effettivamente passati al programma; anche il nome stesso del programma (nell'esempio, fex) e` considerato un argomento, quindi argc ha un valore maggiore o uguale a 1.
- **char **argv:** vettore di stringhe, ciascuna delle quali contiene un diverso argomento. Per convenzione, **argv[0]** contiene il **nome del programma stesso**.

Esempio

```
// sorgente di file.c
main(int argc, char *argv[])
{ ... }
```

- invoco l'eseguibile generato coi seguenti parametri:

- : \$./fex roma 1 x

- quindi:

argc = 4

argv[0] = "./fex"

argv[1] = "roma"

argv[2] = "1"

argv[3] = "x"

NB: gli argomenti sono passati tutti come stringhe

Seconda Esercitazione

Gestione di processi in Unix
Primitive Fork, Wait, Exec

System call fondamentali

fork	<ul style="list-style-type: none">Generazione di un processo figlio, che condivide il codice con il padre e eredita copia dei dati del padreRestituisce il PID (>0) del processo creato per il padre, 0 per il figlio, o un valore negativo in caso di errore
exit	<ul style="list-style-type: none">Terminazione di un processoAccetta come parametro lo stato di terminazione ($0-255$). Per convenzione 0 indica un'uscita con successo, un valore <i>non-zero</i> indica uscita con fallimento.
wait	<ul style="list-style-type: none">Chiamata bloccante.Raccoglie lo stato di terminazione di un figlioRestituisce il PID del figlio terminato e permette di capire il motivo della terminazione (es. volontaria? con quale stato? Involontaria? A causa di quale segnale?)
exec	<ul style="list-style-type: none">Sostituzione di codice (e dati) del processo che l'invocaNON crea processi figli

Esempio - fork e exit

- Consideriamo un programma in cui il processo padre procede alla creazione di un numero N di figli

./generate <N> <term>

Dove :

- N è il numero di figli
- term è un flag [0,1]
 - se 1, ogni figlio fa exit()
 - altrimenti no.

Esempio - Il Codice

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

Simulazione di Esecuzione (1/7)

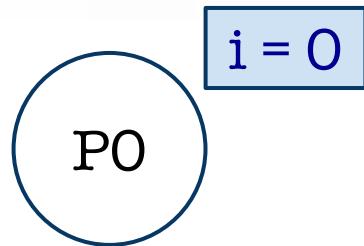
- Vediamo cosa succede durante l'esecuzione del programma
- Assumiamo:
N = 2 : Il padre genera due processi figli
term = '0' : I figli non chiamano exit
- Da ricordare:
Una volta creato, ogni figlio esegue **concorrentemente** al padre e ai fratelli a partire dall'istruzione successiva alla fork() che l'ha creato.

Processo PO

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    →for ( i=0; i<n_children; i++ ) {
        pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 0

Simulazione di Esecuzione (2/7)



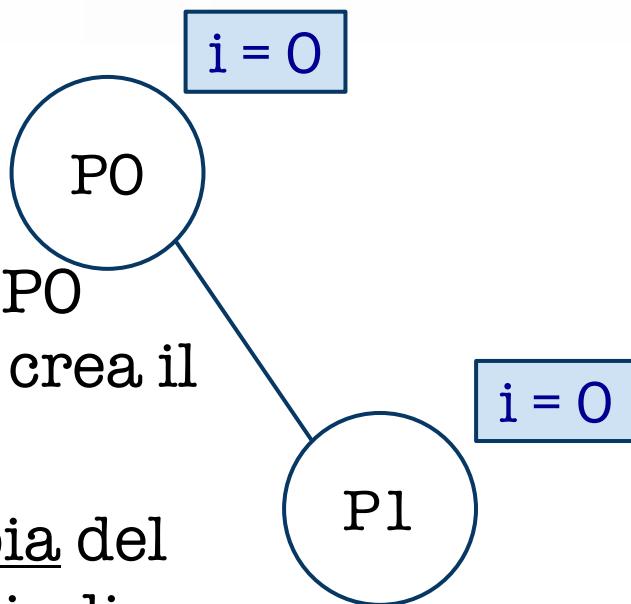
Il processo padre P0 viene
creato e inizia la prima
iterazione del for ($i=0$)

Processo PO

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        → pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 0

Simulazione di Esecuzione (3/7)



Il processo padre P0 esegue la `fork()` e crea il primo figlio P1.

P1 riceve una copia del contesto di P0, quindi anche una sua variabile **i** inizializzata a 0.

Continuiamo a concentrarci su **P0** (padre)

Per il momento trascuriamo **P1**, che intanto sta **eseguendo...**

Processo PO

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            → printf("%d: child created with PID %d\n",
                      getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 0

La prima differenza tra i contesti di PO e P1 è la variabile **pid**.

- **P1: pid=0**
 - **PO: pid>0** (pid del figlio)
- PO esegue la **printf**

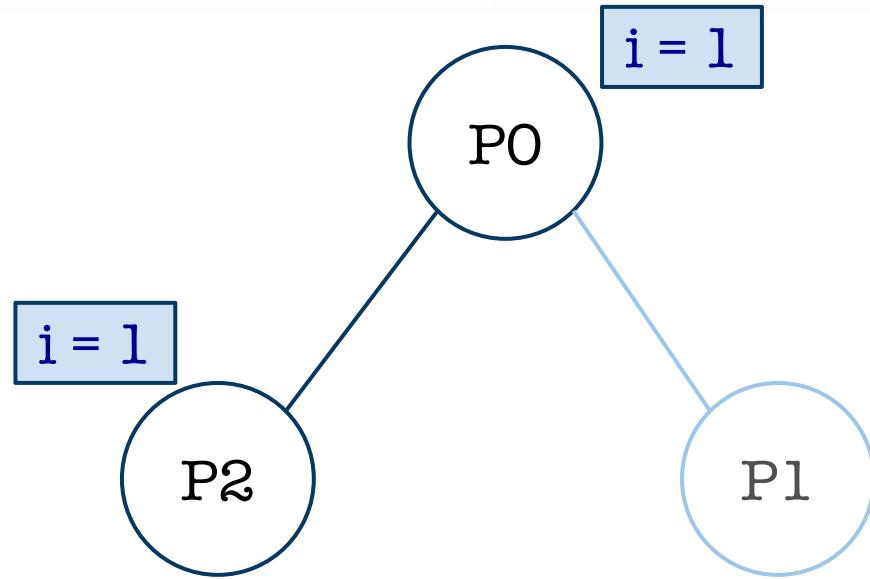
Processo PO

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        → pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' ) exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 1

PO continua l'esecuzione e ricomincia il ciclo for con **i=1**. Esegue ancora una fork

Simulazione di esecuzione (4/7)



La fork eseguita da P0 genera P2, che riceve una copia del contesto di P0. Quindi P2 riceve anche una variabile **i** inizializzata a 1.

Processo PO

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            → printf("%d: child created with PID %d\n",
                      getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 1

PO esegue ancora una
printf()

Processo PO

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    →for ( i=0; i<n_children; i++ ) {
        pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 2

PO ricomincia il ciclo for:
i=2. Testa la condizione
(2<2), esce dal for

Simulazione di esecuzione (5/7)

P0 a questo punto ha creato tutti i figli che doveva

MA

Cosa hanno fatto i suoi figli nel frattempo ?

Iniziamo da **P2**...

Ricordate: i processi figli non terminano subito dopo essere stati creati (term = '0')

Processo P2

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        pid = fork();
        → if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 1

P2 esegue il suo codice a partire da if(pid==0)

Processo P2

```
void main(int argc, char *argv[]) {  
    int i, j, k, pid, status, n_children;  
    char term;  
    n_children = atoi(argv[1]);  
    term = argv[2][0];  
    →for ( i=0; i<n_children; i++ ) {  
        pid = fork();  
        if ( pid == 0 ) { // Eseguito dai figli  
            if ( term == '1' ) exit(0);  
        }  
        else if ( pid > 0 ) { // Eseguito dal padre  
            printf("%d: child created with PID %d\n",  
                   getpid(), pid);  
        }  
        else {  
            perror("Fork error:");  
            exit(1);  
        }  
    }  
}
```

i = 2

P2 ricomincia il ciclo for:
i=2. Testa la condizione
(2<2), esce dal for e
termina.

Simulazione di esecuzione (..continua)

Analizziamo il comportamento di **P1**....

Processo P1

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        pid = fork();
        → if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 0

P1 esegue il suo codice a partire da if(pid==0)

Processo P1

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    →for ( i=0; i<n_children; i++ ) {
        pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 1

Poichè la sua copia di i vale 0, P1 ricomincia il ciclo con i=1.

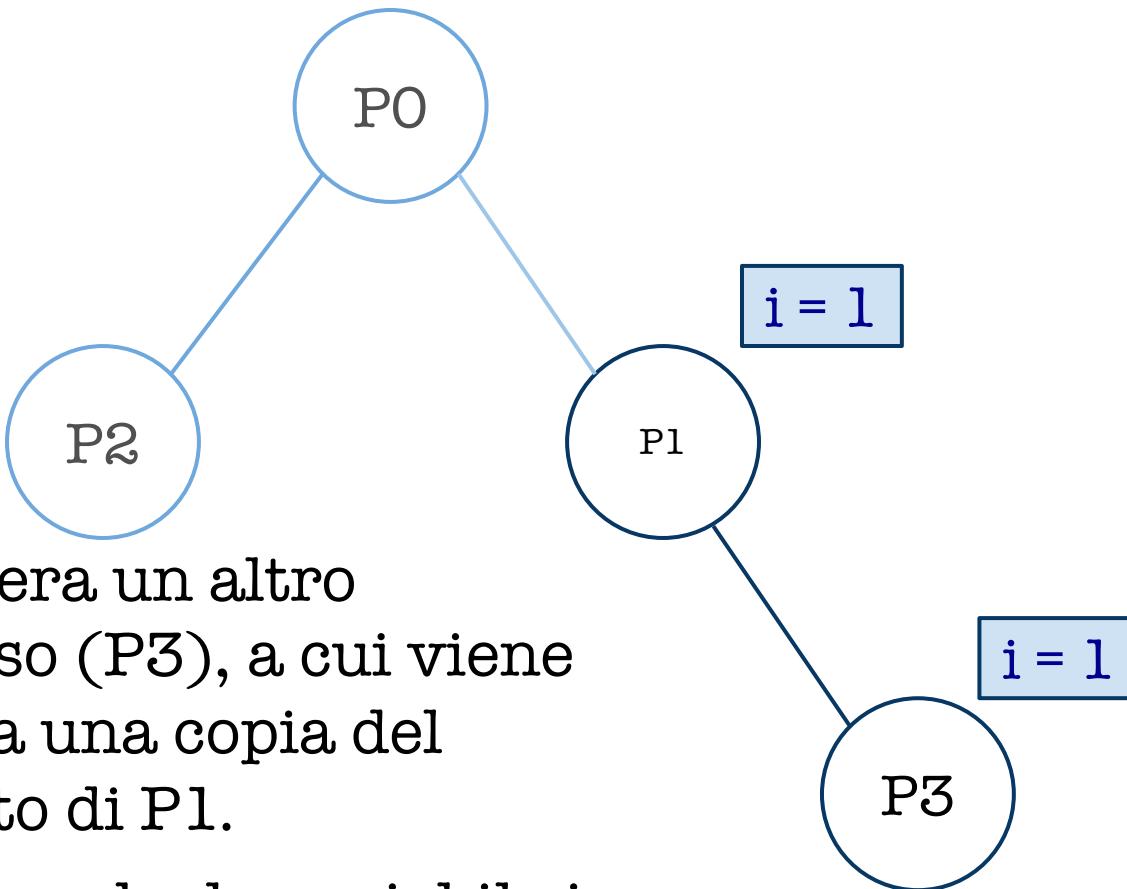
Processo P1

```
void main(int argc, char *argv[]) {
    int i, j, k, pid, status, n_children;
    char term;
    n_children = atoi(argv[1]);
    term = argv[2][0];
    for ( i=0; i<n_children; i++ ) {
        → pid = fork();
        if ( pid == 0 ) { // Eseguito dai figli
            if ( term == '1' )      exit(0);
        }
        else if ( pid > 0 ) { // Eseguito dal padre
            printf("%d: child created with PID %d\n",
                   getpid(), pid);
        }
        else {
            perror("Fork error:");
            exit(1);
        }
    }
}
```

i = 1

P1 esegue un'altra fork!

Simulazione di esecuzione (6/7)



P1 genera un altro processo (P3), a cui viene passata una copia del contesto di P1.

Quindi anche la variabile i inizializzata a 1

Morale

- Quando si usa la *system call* **fork()**, bisogna sempre tener presente che i dati del processo padre vengono duplicati nel processo figlio e che la sua esecuzione prosegue secondo quanto descritto nel codice (almeno inizialmente condiviso) del programma.
 - Trascurare questo "dettaglio" può portare a comportamenti indesiderati
-

Sesta Esercitazione

File
comandi
Unix



Esempio di file comandi

Scrivere un file comandi da invocare come segue:

./esempio D

dove D è il nome di una directory esistente.

Dopo un opportuno controllo sugli argomenti, lo script dovrà controllare ogni 5 secondi se sono stati **creati o eliminati file nella directory D**:

- In caso di cambiamento, si deve **visualizzare un messaggio su stdout** che comunichi quanti file sono presenti nella directory.

Suggerimento: uso di un file temporaneo, in cui tenere traccia del numero di file presenti al controllo precedente

Esempio: soluzione

```
#!/bin/bash
if [ $# -ne 1 ] ; then echo Sintassi! ; exit; fi
if [ -d $1 ] ; then echo $1 è una directory esistente
else echo $1 non è un dir! ; exit; fi
echo 0 > loop.$$.tmp
OK=0
while [ $OK -lt 10 ]
do
    new=`ls "$1" | wc -w`
    old=`cat loop.$$.tmp`  

    if [ $new -ne $old ]
    then
        echo $new > loop.$$.tmp
        echo in $1 ci sono $new file
    fi
    OK=`expr $OK + 1`
    sleep 5s
done
rm loop.$$.tmp
```

numero di parametri, \$0 escluso

pid del processo in esecuzione

"" evitano problemi in caso di
parametro \$1 con spazi

i nomi di file in \$1 potrebbero
contenere spazi. Meglio:
new=`ls -1 "\$1" | wc -l`
new=`expr \$new - 1`

Esercizio 1

Creare uno script che abbia la sintassi

./elabora S

dove S è una stringa di caratteri.

Lo script deve:

- richiedere all'utente e **leggere da standard input** il path assoluto di un file F.
- **controllare** che F sia un path assoluto e corrisponda al nome di un file esistente e leggibile.
- scrivere in un **file di output** le linee del file F che contengono almeno una occorrenza della stringa S , **ordinate in ordine lessicografico inverso**.

Il file di output sarà memorizzato nella home directory dell'utente che ha invocato lo script e dovrà avere il nome:

results_<uname>.out

dove <uname> è il nome dello USER che ha invocato lo script

Esercizio 1: suggerimenti

Lettura da standard input:

- **read var1 var2**

Le stringhe in ingresso vengono attribuite alle variabili a seconda della corrispondenza posizionale

Test di file:

- **test -f <path>** Esistenza del file. Alternativa [**-f <path>**]
- **test -d <path>** Esistenza del directory
- **test -r <path>** Diritto di lettura (allo stesso modo, **-w e -x**)

Ricerca di una stringa in un file:

- **grep** (default case sensitive) → quale opzione per comportamento case insensitive? (v. man)

Ordinamento delle linee di un file:

- **sort** → quale opzione per ordinamento inverso? (v. man)
[perchè sort e non rev?]

Redirezione I/O

- **comando > F** st. output redirezionato sul file F path(>> per append)
- **comando < F** st. input preso dal file F

L'output del grep deve essere elaborato dal sort → **piping di comandi**

Esercizio 2

Realizzare un file comandi che preveda la seguente sintassi:

cerca S M D1 D2 .. DN

dove:

- **S** è una stringa
- **M** è un intero
- **D1, D2, DN** sono nomi assoluti di directory esistenti.

Il file comandi deve:

- **controllare** il corretto passaggio degli argomenti;
- **ispezionare** il contenuto di tutte le directory date (**D1, D2, .. DN**) allo scopo di **individuare tutti file che contengono esattamente M occorrenze della stringa S**.
- Il file comandi dovrà **stampare** a video il nome assoluto di ogni file con tali caratteristiche e, al termine, stampare il numero totale dei file individuati.

Esercizio 2: suggerimenti

Ciclo su un elenco di directory con path assoluto:

```
for dir in /path/to/dir1 /path/to/dir2 /path/to/dir3  
do  
    # do something on $dir  
done
```

L'esercizio richiede di iterare su un elenco di directory fornite dalla linea di comando: quale **variabile notevole** devo utilizzare?

Come calcolare il numero di occorrenze di una stringa in un file?

Vedere **grep -o ...** e **wc -w** (consultare il man)

Altri suggerimenti

Provare i comandi a linea di comando prima di scriverli nello script bash!

posso provare i comandi semplici:

```
studente@debian:~$ grep -c file1.txt
```

ma anche i comandi più complessi come condizioni, if e cicli:

```
studente@debian:~$ if test -f pippo ; then echo  
yes ; else echo no; fi
```

```
studente@debian:~$ for fname in *; do echo  
$fname ; done
```

**Ulteriore esercizio
per continuare a casa..**



Esercizio 3

Creare uno script che abbia la sintassi

`./conteggio F USER filedir`

Dove: **F** è il nome relativo di un file esistente, **USER** è uno username, **filedir** è il nome assoluto di un file leggibile esistente contenente una serie di nomi assoluti di directory esistenti.

Lo script dovrà cercare **nelle directory elencate in filedir tutti file di nome F di proprietà dell'utente U**; per ogni file che soddisfa questa condizione, lo script dovrà calcolarne la dimensione in bytes e stampare la stringa seguente:

«Il file <nome file> dell'utente U nella directory <Di> contiene <dim> caratteri.»

Esercizio 3: suggerimenti

Ciclo su un elenco di directory contenute in un file:

- ricordiamo che il comando cat stampa il contenuto del file dato come arg.
- ricordiamo il significato dei backquote: `cat FILEDIR`

Come estrarre l'username del proprietario di un file?

- ricordiamo il comando cut applicato all'output di **ls -l**

```
-rw-r--r-- 1 anna staff 2717 15 Apr 10:16 esempio5.c
```



lo username è il terzo “campo” → **cut** con opzione **-f3**

Settima esercitazione

Shell scripting

Agenda

Esempio

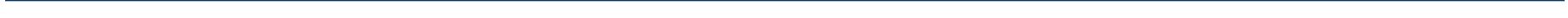
Esplorazione completa di una directory: script bash con ricorsione.

Esercizio 1

Esplorazione ricorsiva del file system

Esercizio 2

Esplorazione ricorsiva di più sottoalberi



Esempio – Script ricorsivi

Si scriva uno script bash avente interfaccia di invocazione

reurse_dir.sh dir

Il programma, dato un directory in ingresso **dir**, deve stampare su stdout l'elenco dei file contenuti nel directory e in tutti i suoi sottodirectory (analogamente al comando **ls -R**)

Schema di soluzione ricorsiva

`recurse_dir.sh arg1`

caso **base**

arg1 è un file

→ stampo il nome

caso generale espresso in termini **ricorsivi**

arg1 è una directory

→ mi muovo nella directory **arg1**

per ogni file (normale o directory) **invoco**
nuovamente `recurse_dir.sh`

Bozza di soluzione

```
#!/bin/bash  
if ! test -d "$1" ; then  
    echo `pwd`/$1
```

Caso
base

```
else  
    cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```

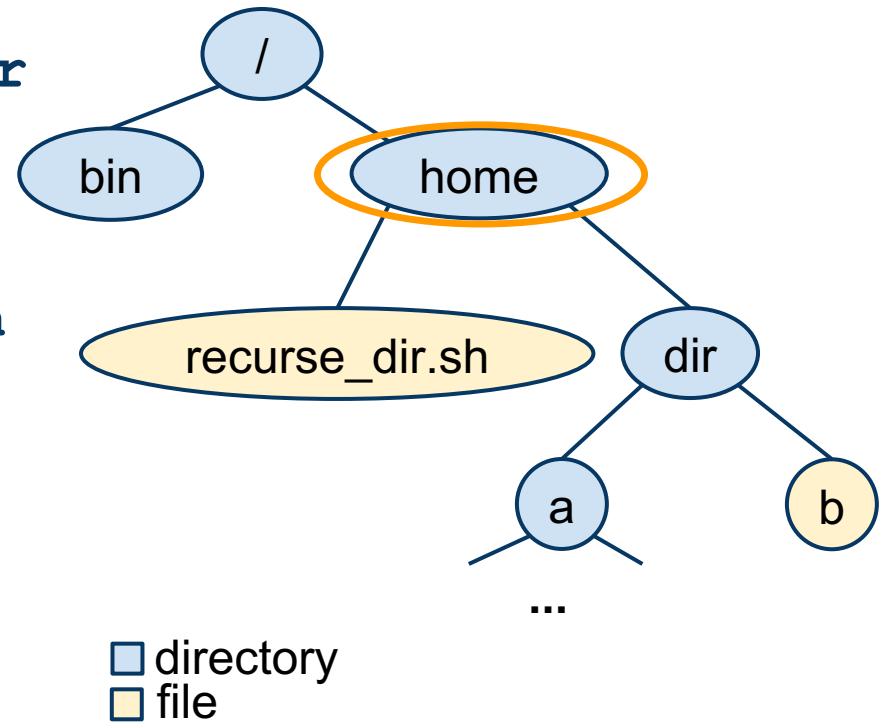
Caso
generale

Chiamata ricorsiva

Ricorsione (1/6)

```
$ pwd  
/home  
$ /home/recuse_dir.sh dir
```

```
if ! test -d "$1" ; then  
    echo `pwd`/$1  
else  
    cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



VARIABILI:

\$PWD /home

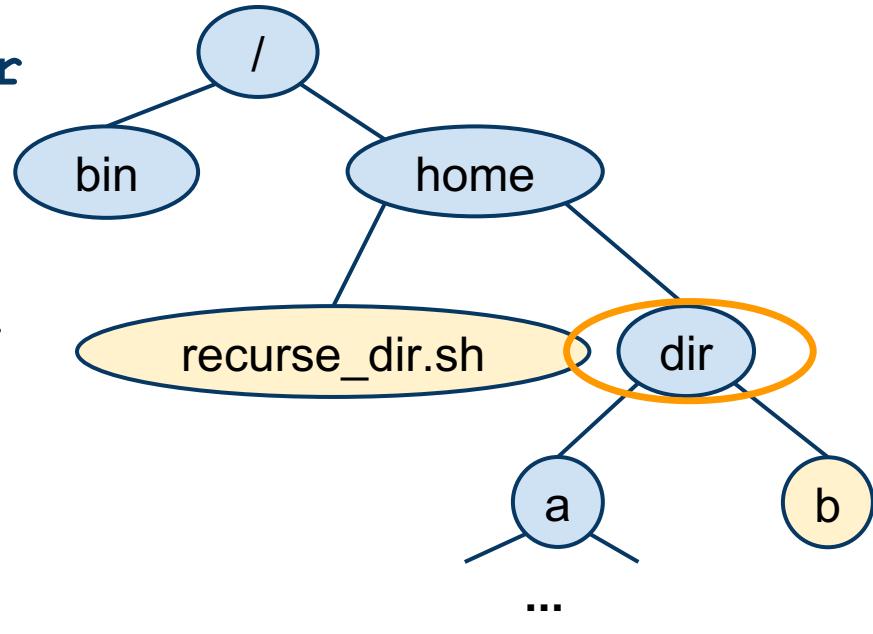
\$0 /home/recuse_dir.sh

\$1 dir

Ricorsione (2/6)

```
$ pwd  
/home  
$ /home/recurse_dir.sh dir
```

```
if ! test -d "$1" ; then  
    echo `pwd`/$1  
else  
    → cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



VARIABILI:

\$PWD /home/dir

\$0 /home/recurse_dir.sh

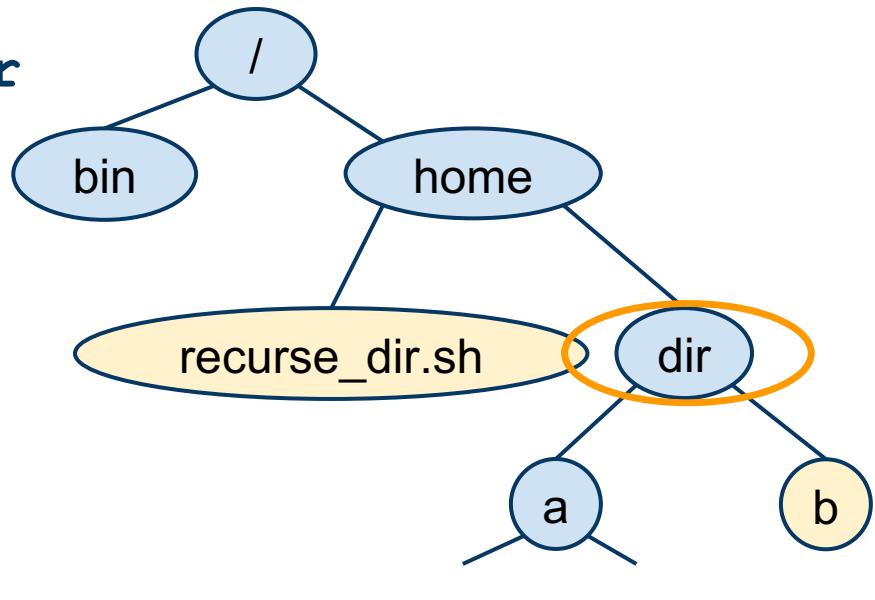
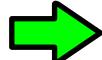
\$1 dir

Ricorsione (3/6)

```
$ pwd  
/home  
$ /home/recurse_dir.sh dir
```

```
$ /home/recuse_dir.sh a
```

```
if ! test -d "$1" ; then  
    echo $PWD/$1  
else  
    cd "$1"  
    for f in * ; do  
        echo "$0" "$f"  
    done  
fi
```



■ directory
■ file

VARIABILI:

\$PWD /home/dir

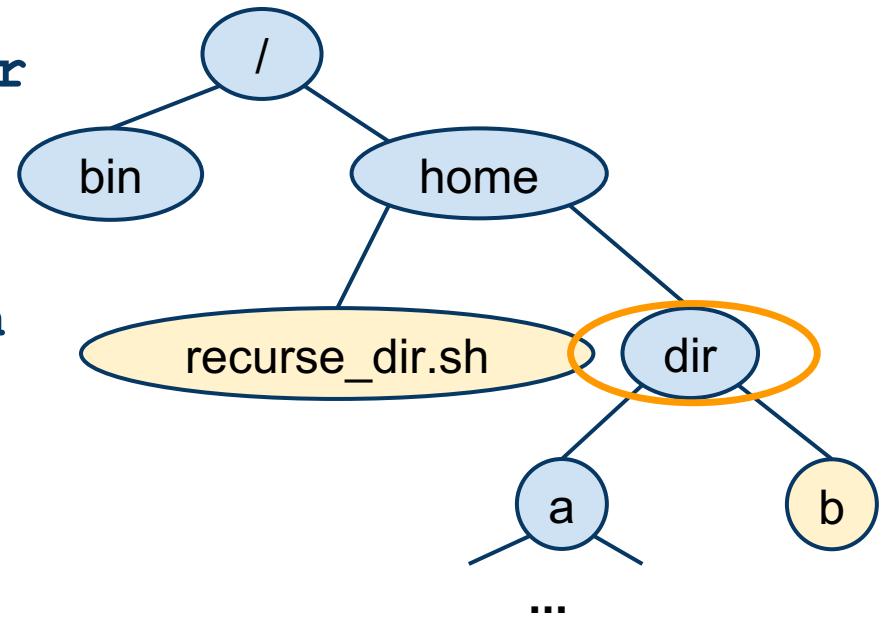
\$0 /home/recuse_dir.sh

\$1 dir

Ricorsione (4/6)

```
$ pwd  
/home  
$ /home/recurse_dir.sh dir
```

```
if ! test -d "$1" ; then  
    echo `pwd`/$1  
else  
    cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



■ directory
■ file

VARIABILI:

\$PWD /home/dir

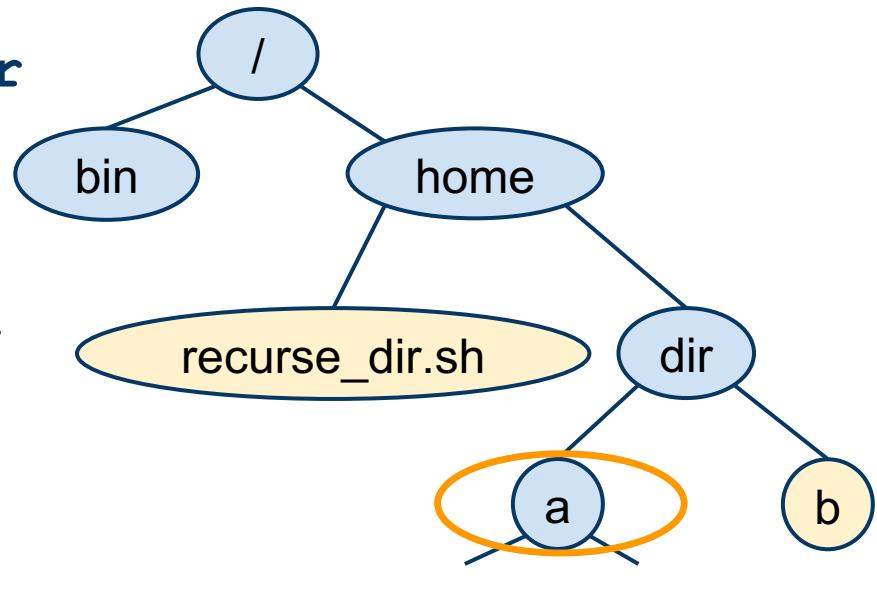
\$0 /home/recurse_dir.sh

\$1 a

Ricorsione (5/6)

```
$ pwd  
/home  
$ /home/recurse_dir.sh dir
```

```
if ! test -d "$1" ; then  
    echo `pwd`/$1  
else  
    cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



■ directory
■ file

VARIABILI:

\$PWD /home/dir/a

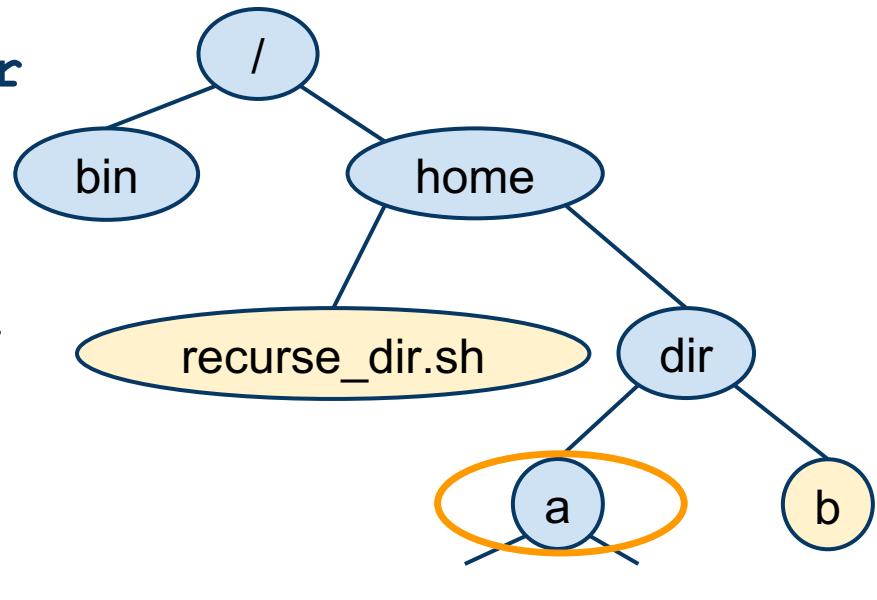
\$0 /home/recurse_dir.sh

\$1 a

Ricorsione (6/6)

```
$ pwd  
/home  
$ /home/recurse_dir.sh dir
```

```
if ! test -d "$1" ; then  
    echo `pwd`/$1  
else  
    cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



■ directory
■ file

VARIABILI:

\$PWD /home/dir/a
\$0 /home/recurse_dir.sh
\$1 ...

ATTENZIONE

Nell'esempio lo script è stato invocato **specificando il suo nome assoluto:**

```
$ /home/recurse_dir.sh dir
```

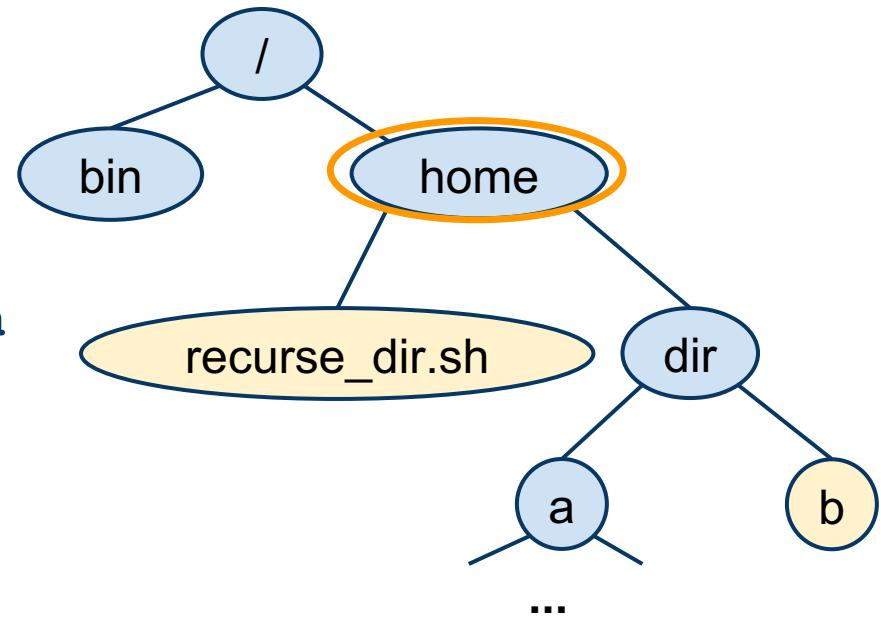
Cosa succederebbe invocandolo
con un **nome relativo?**

```
$ ./recurve_dir.sh dir
```

Ricorsione - alternativa (1/3)

```
$ pwd  
/home  
$ ./recurse_dir.sh dir
```

```
if ! test -d "$1" ; then  
    echo `pwd`/$1  
else  
    cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



■ directory
■ file

VARIABILI:

\$PWD /home

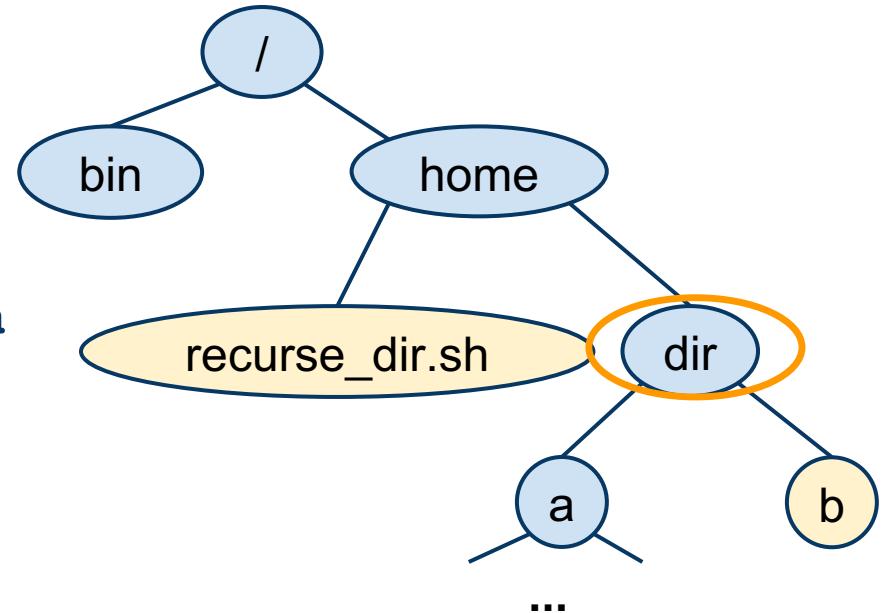
\$0 ./recurse_dir.sh

\$1 dir

Ricorsione - alternativa (2/3)

```
$ pwd  
/home  
$ ./re recurse_dir.sh dir
```

```
if ! test -d "$1" ; then  
    echo `pwd`/$1  
else  
    ➔ cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



■ directory
■ file

VARIABILI:

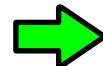
\$PWD /home/dir
\$0 ./re recurse_dir.sh
\$1 dir

Ricorsione - alternativa (3/3)

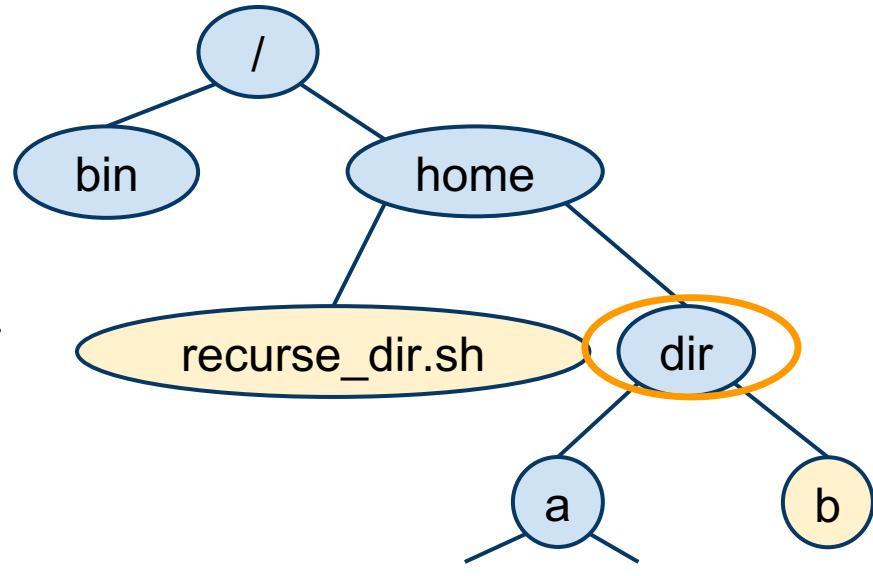
```
$ pwd  
/home  
$ ./recurse_dir.sh dir
```

```
$ ./recurse_dir.sh a
```

```
if ! test -d "$1" ; then  
    echo $PWD/$1  
else  
    cd "$1"  
    for f in * ; do  
        "$0" "$f"  
    done  
fi
```



non funziona!



■ directory
■ file

VARIABILI:

\$PWD /home/dir
\$0 ./recurse_dir.sh
\$1 dir



Come risolvere?

Problema: Un valore dipendente dalla directory di lavoro corrente (un percorso relativo) viene "**propagato**" da una invocazione ricorsiva all'altra (tramite la variabile **\$0**)

La directory di lavoro però cambia (perchè usiamo il comando **cd** nel codice)

Possibile soluzione: Prima di iniziare la ricorsione memorizzare la directory di partenza in una variabile che verrà usata per le invocazioni ricorsive

Occorre creare:

- **Script ricorsivo**
- **Script di invocazione:**

Controlla i parametri

Salva in maniera "stabile" il percorso dello script ricorsivo
Innesca la ricorsione

Struttura di un file comandi ricorsivo

invoker.sh

```
#!/bin/sh
```

Controllo degli argomenti

Invocazione del file comandi ricorsivo do_recursive.sh

do_recursive.sh

```
#!/bin/sh
```

Esecuzione del compito

Invocazione del file comandi ricorsivo do_recursive.sh

Script di invocazione

re recurse_dir.sh

```
#!/bin/bash
# ... controllo argomenti

oldpath=$PATH
PATH=$PATH:`pwd`  
do_recurse_dir.sh "$1"
PATH=$oldpath
```

do_recurse_dir.sh

```
#!/bin/bash
if ! test -d "$1" ; then
    echo `pwd`/$1
else
    cd "$1"
    for f in * ; do
        "$0" "$f"
    done
fi
```

PATH è una variabile d'ambiente che contiene dei nomi di directory separati da ":".

Quando lancio un comando senza alcun path (né assoluto né relativo, es: invoco **ls** invece di **/bin/ls**), il SO cerca quel comando in tutte le directory contenute nella variabile **PATH**.

Script di invocazione

re recurse_dir.sh

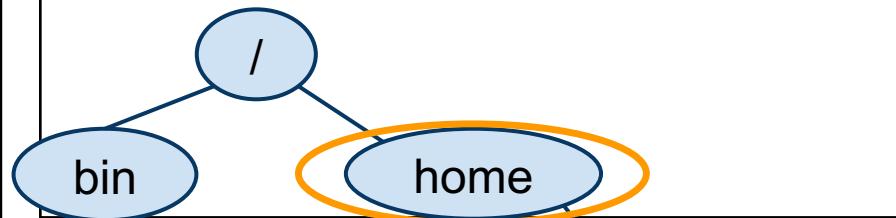
```
#!/bin/bash
# ... controllo argomenti

oldpath=$PATH
PATH=$PATH:`pwd`  

do_recurse_dir.sh "$1"
PATH=$oldpath
```

do_recurse_dir.sh

```
#!/bin/bash
if ...
```



Problema :

Che succede se gli script si trovano in
/home/dloreti e l'utente li invoca dalla directory corrente
/home con il path relativo: **./dloreti/recurse_dir.sh**
=> `pwd` viene espanso in “**/home**”
=> **PATH=\$PATH:/home**
=> **do_recurse_dir.sh** viene cercato in **/home** → **NON TROVATO!**

Soluzione generale

recurse_dir.sh

```
#!/bin/bash
# ... controllo argomenti

if [[ "$0" = /* ]] ; then
    # se $0 è un path assoluto
    dir_name=`dirname "$0"`
    recursive_cmd="$dir_name/do_recurse_dir.sh"
elif [[ "$0" = */* ]] ; then
    # se c'è uno slash, ma non inizia con /
    # $0 è un path relativo
    dir_name=`dirname "$0"`
    recursive_cmd=`pwd`/$dir_name/do_recurse_dir.sh"
else
    # Non si tratta né di un path relativo, né di uno
    # assoluto, il comando $0 sarà cercato in $PATH.
    recursive_cmd=do_recurse_dir.sh
fi
#Invoco il comando ricorsivo
"$recursive_cmd" "$1"
```

Restituisce \$0 tranne
l'ultimo / e ciò che segue

do_recurse_dir.sh

```
#!/bin/bash
if ...
```

Esercizio 1 - Esplorazione ricorsiva del file system (1/2)

Realizzare un file comandi (ricorsivo) che abbia la sintassi

cerca U R dir

dove:

U è una stringa che rappresenta lo **username** di un utente

R è un **intero** positivo

dir è il nome **assoluto** di un direttorio esistente nel file system

Esercizio 1 - (2/2)

Il compito del file comandi è quello di :

- Esplorare (ricorsivamente) il sottoalbero individuato da **dir**
- Individuare i file **ordinari e leggibili di proprietà dell'utente U**
- Per ogni file che rispetti tali caratteristiche **stampare** a video **il suo nome assoluto** e le **ultime R righe** contenute in esso.

Esercizio 1 : suggerimenti

Prima di tutto realizzare la ricorsione e testare che funzioni correttamente!

Poi:

- Individuare lo username del proprietario di un file → vedere il comando **stat** per ottenere gli attributi di un file (**man stat**):
→ per lo **username**: opzione **--format=%U**
- Stampare le ultime R righe di un file → vedere il comando **tail** con opzione **-n** (**man tail**)

Esercizio 2 - Esplorazione ricorsiva di N directory (1/2)

Realizzare un file comandi (ricorsivo) che abbia la sintassi

cerca Fout U R dir1..dirN

dove:

Fout è il nome **assoluto** di un file inizialmente non presente nel file system;

U è una stringa che rappresenta lo **username** di un utente;

R è un **intero** positivo;

dir1..dirN sono nomi **assoluti** di direttori esistenti nel file system

Esercizio 2 - (2/2)

Il compito del file comandi è quello di eseguire una ricerca in tutte le directory **dir₁...dir_N**.

Per ogni directory **dir_i**:

- Esplorare (ricorsivamente) il sottoalbero individuato da **dir_i** allo scopo di individuare i file **ordinari e leggibili di proprietà dell'utente U**;
- Per ogni file che rispetti tali caratteristiche **aggiungere** al file **Fout** **il suo nome assoluto** e stampare a video le **ultime R righe** contenute in esso.

Al termine della ricerca su tutte le directory, il file comandi dovrà **stampare sullo standard output il numero di file** (che rispettano le caratteristiche date) **trovati nelle directory esplorate**.

Suggerimenti

- Invoker deve iterativamente attivare lo script ricorsivo sulle directory date. Ricordare:
 - `$@` (o `$*`) → lista delle variabili posizionali
 - `shift` → scorrimento a sinistra delle var posizionali

```
shift  
shift  
shift  
for d in $@  
do ... done
```

- alla fine l'invoker deve ricavare il numero totale dei file trovati → conteggio delle linee di Fout

Suggerimenti

- Invoker deve iterativamente attivare lo script ricorsivo sulle directory date. Ricordare:
 - `$@` → lista delle variabili posizionali
 - `shift` → scorrimento a sinistra delle var posizionali

```
shift  
shift  
for d in $@  
do ... done
```

Ottava Esercitazione

Gruppo A-K

Introduzione ai thread java
mutua esclusione



Agenda

I thread in java: creazione e attivazione di threads.

Esempio

- Concorrenza in Java: creazione ed attivazione di thread concorrenti.

Mutua esclusione in java: metodi synchronized

Esercizio 1 - da svolgere

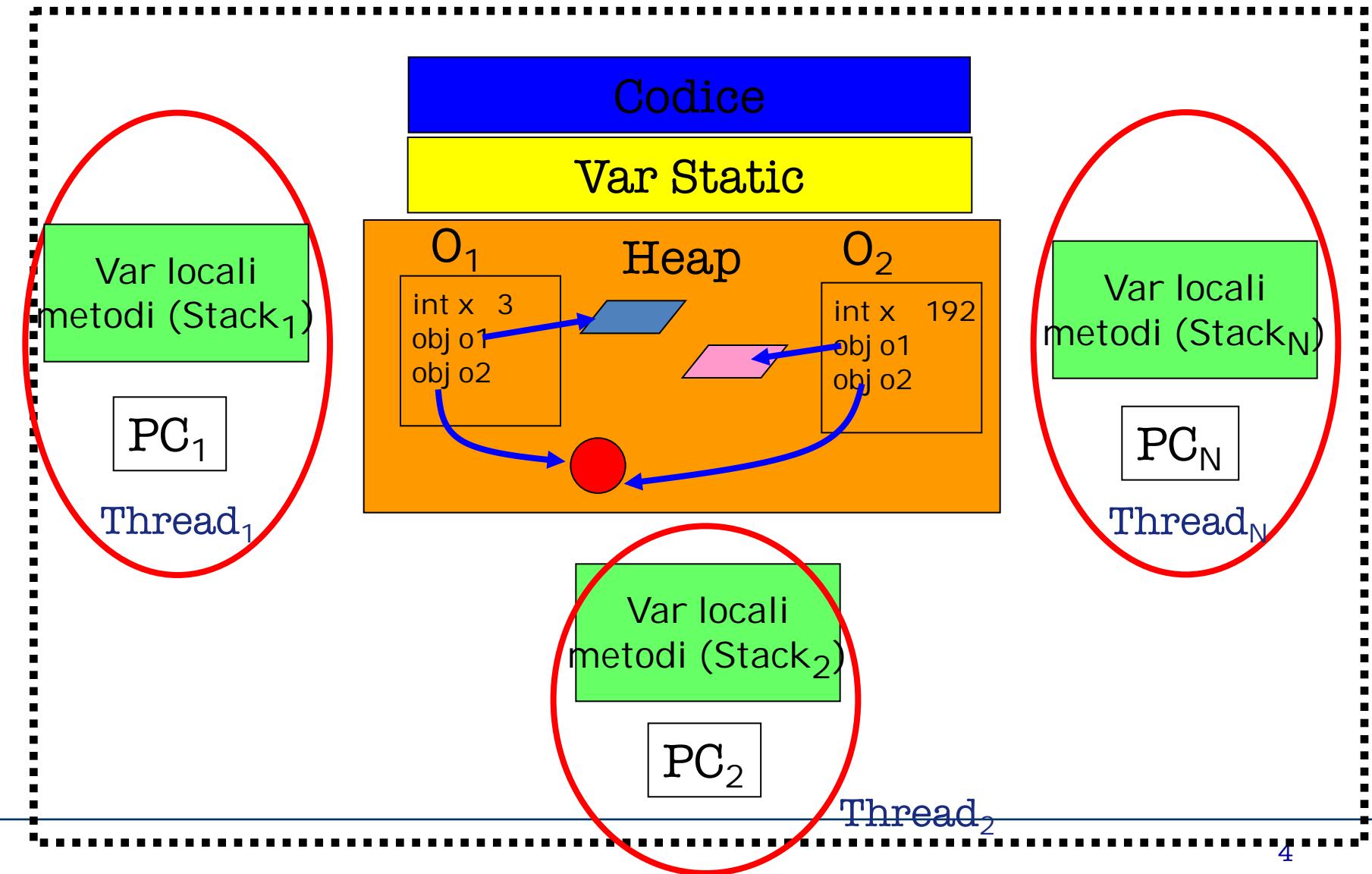
- Concorrenza in Java: sincronizzazione di thread concorrenti tramite synchronized

I threads in Java

- All'esecuzione di ogni programma Java corrisponde un task che contiene almeno un singolo thread, corrispondente all'esecuzione del metodo main() sulla JVM.
- E' possibile creare dinamicamente nuovi thread attivando concorrentemente le loro esecuzioni all'interno del programma.

Java Thread

Processo



Java Thread: programmazione

Due metodi per definire thread in Java:

1. estendendo la classe Thread

2. implementando l' interfaccia Runnable

Primo metodo: esempio

```
public class SimpleThread extends Thread{  
  
    public SimpleThread(String str)  
    {super(str);}  
  
    public void run() {  
        for(int i=0; i<10; i++)  
        { System.out.println(i+ " " +getName());  
        try{  
            sleep((int)Math.random()*1000);  
        } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! "+getName());  
    }  
}
```

```
public class EsempioConDueThreads
{
    public static void main(String[] args)
    { SimpleThread st1= new SimpleThread("Pippo");
      st1.start();
    }
}
```

2. Thread come classi che implementano Runnable

Se un thread deve ereditare da una classe C diversa da Thread, la definizione si ottiene

1. Definendo una nuova classe **C1** che:
 - **estende** la classe C
 - **implementa** l'interfaccia **Runnable**
 - **ridefinisce** il metodo **run()**.
2. si crea un'istanza **o1** della classe **C1** tramite **new**;
3. si crea **un'istanza t della classe Thread** con **new**, passandole come **parametro** l'oggetto **o1** **creato al punto 2** ;
4. si esegue il thread invocando il metodo **start()** **sull'oggetto t** **creato al punto 3**.

Secondo metodo: esempio

```
class C1 extends C implements Runnable
{
    public void run() {
        for (int i=1; i<=10; i++)
            System.out.println(i + " " + i*i);
    }
}

public class Esempio {
    public static void main(String args[]) {
        C1 o1 = new C1();
        Thread t = new Thread (o1);
        t.start();
    }
}
```

Esempio - Traccia (2/2)

Il programma deve definire le seguenti classi:

- **Automobile**: definisce le caratteristiche comuni di un'auto (marca, modello, targa, cilindrata, ...), un metodo astratto **getType()** ed un metodo concreto **getMessage()** che ritorna il messaggio da stampare e che richiami **getType()** per capire il tipo di automobile.
- **AutoGrande**: eredita da Automobile e specializza il comportamento di **getType()** in modo che ritorni la stringa "auto grande"
- **AutoPiccola**: eredita da Automobile e specializza **getType()** in modo che ritorni la stringa "auto piccola"
- **Moto**: definisce le caratteristiche della moto
- **Autolavaggio**: implementa il metodo **main()** che crea un numero (a scelta) di veicoli di ciascun tipo e li mette in esecuzione tramite thread

Esempio – Definizione e uso dei java thread

Scrivere una applicazione Java che simuli un autolavaggio.

- Nell'autolavaggio possono entrare sia automobili che moto.
- Le **automobili** possono essere di due tipi, ossia auto **grandi** oppure auto **piccole**.
- Le **moto**, invece, sono di un unico tipo.

Tutti gli autoveicoli devono essere **oggetti attivi** (ossia in grado di eseguire in maniera concorrente tramite thread)

In particolare, ciascun autoveicolo, quando eseguito, dovrà **stampare su stdout** un opportuno messaggio che descriva le sue caratteristiche.

Soluzione: classe Automobile

```
public abstract class Automobile {  
  
    private String marca;  
    private String modello;  
    private String targa;  
    private int cilindrata;  
  
    public Automobile(String marca, String modello,  
                      String targa, int cilindrata){  
        this.marca = marca;  
        this.modello = modello;  
        this.cilindrata = cilindrata;  
        this.targa = targa;  
    }  
// continua..
```

..classe Automobile

```
//... continua  
public abstract String getType();  
  
public String getMessage(){  
    return this + " Tipo " + getType() +  
        " Marca " + this.marca + "; Modello " +  
        this.modello + "; Cilindrata " +  
        this.cilindrata;  
}  
  
public String toString(){  
    return "[Automobile con targa: " +  
        this.targa +"]";  
}  
  
}//end of class Automobile
```

Soluzione: classe AutoGrande

```
public class AutoGrande extends Automobile  
    implements Runnable {
```

Non posso estendere Thread, perchè devo già estendere Automobile. Quindi implemento Runnable.

```
public AutoGrande(String marca, String modello,  
                  String targa, int cilindrata)  
{ super(marca, modello, targa, cilindrata); }
```

```
@Override  
public String getType()  
{ return "AutoGrande"; }
```

// continua..

..classe AutoGrande

//... continua

@Override

public void run() {

System.out.println("Il thread per l'automobile "
+ this + " ha iniziato" +"l'esecuzione.");

System.out.println(this.getMessage());

System.out.println("Il thread per l'automobile "
+ this + " sta per terminare");

}

Soluzione: classe AutoPiccola

```
public class AutoPiccola extends Automobile
    implements Runnable {

    public AutoPiccola(String marca, String
modello,           String targa, int cilindrata)
    { super(marca, modello, targa, cilindrata);}

    @Override
    public String getType() {
        return "AutoPiccola";
    }

    // continua..
```

..classe AutoPiccola

```
// ..continua  
@Override  
public void run() {  
    System.out.println("Il thread per  
    l'automobile "+this+" ha iniziato"  
    +"l'esecuzione.");  
    System.out.println(this.getMessage());  
    System.out.println("Il thread per  
    l'automobile "+this+" sta per terminare");  
}  
  
}//end of class AutoPiccola
```

Soluzione: classe Moto

```
public class Moto extends Thread {  
    private String marca;  
    private String targa;  
    private String modello;  
    private int cilindrata;  
  
    public Moto(String marca, String modello,  
               String targa, int cilindrata) {  
        this.marca = marca; this.targa = targa;  
        this.modello = modello;  
        this.cilindrata = cilindrata;  
    }  
    public String getMessage() {  
        return this+" Marca "+this.marca+"; Modello "  
               +this.modello+"; Cilindrata  
               "+this.cilindrata;  
    }  
    // continua..
```

Posso estendere Thread, perchè questa classe non deve estendere nient'altro

..classe Moto

```
//.. continua  
@Override  
public String toString() {  
    return "[Moto con targa: " + this.targa + "]";  
}  
  
@Override  
public void run() {  
    System.out.println("Il thread per la moto " +  
        this + " ha iniziato" + "l'esecuzione.");  
    System.out.println(this.getMessage());  
    System.out.println("Il thread per la moto " +  
        this + " sta per terminare");  
}  
} //end of class Moto
```

Classe Autolavaggio

```
public class AutoLavaggio{  
    public static void main(String[] args) {  
        AutoPiccola a1 = new AutoPiccola("FIAT",  
                                         "Modello1", "AB123BC", 2000);  
        AutoGrande a2 = new AutoGrande("Mercedes",  
                                         "Modello2", "ILNY", 3000);  
        Moto m1 = new Moto("Kawasaki", "Ninja",  
                           "ASTFG", 2);  
        Thread t1 = new Thread(a1);  
        Thread t2 = new Thread(a2);  
        t1.start();  
        t2.start();  
        m1.start();  
    }  
}
```

AutoPiccola e AutoGrande non sono Thread, implementano solo Runnable. Mi devo solo ricordare di metterle in un Thread per poterle far partire.

Note

Provare l'applicazione (download dal sito web del corso).

Due versioni:

- **SimpleLavaggio**
- **RandomLavaggio** (definizione casuale di tipologia e numero dei thread da generare, v. `java.util.Random`)

Link utili:

- Oracle Java Doc per Java 8
SE: <http://docs.oracle.com/javase/8/docs/api/>
 - Buon tutorial Oracle sulla concorrenza in Java: <http://docs.oracle.com/javase/tutorial/essential/concurrency/>
-

Attesa teminazione thread: join()

Un thread può sincronizzarsi con la terminazione di altri thread utilizzando il metodo join():

```
T.join();
```

dove T è il thread del quale si attende la terminazione.

Es:

```
Mythread T=new Mythread();  
T.start();  
...  
T.join(); // il thread corrente attende la  
// terminazione del thread figlio T
```

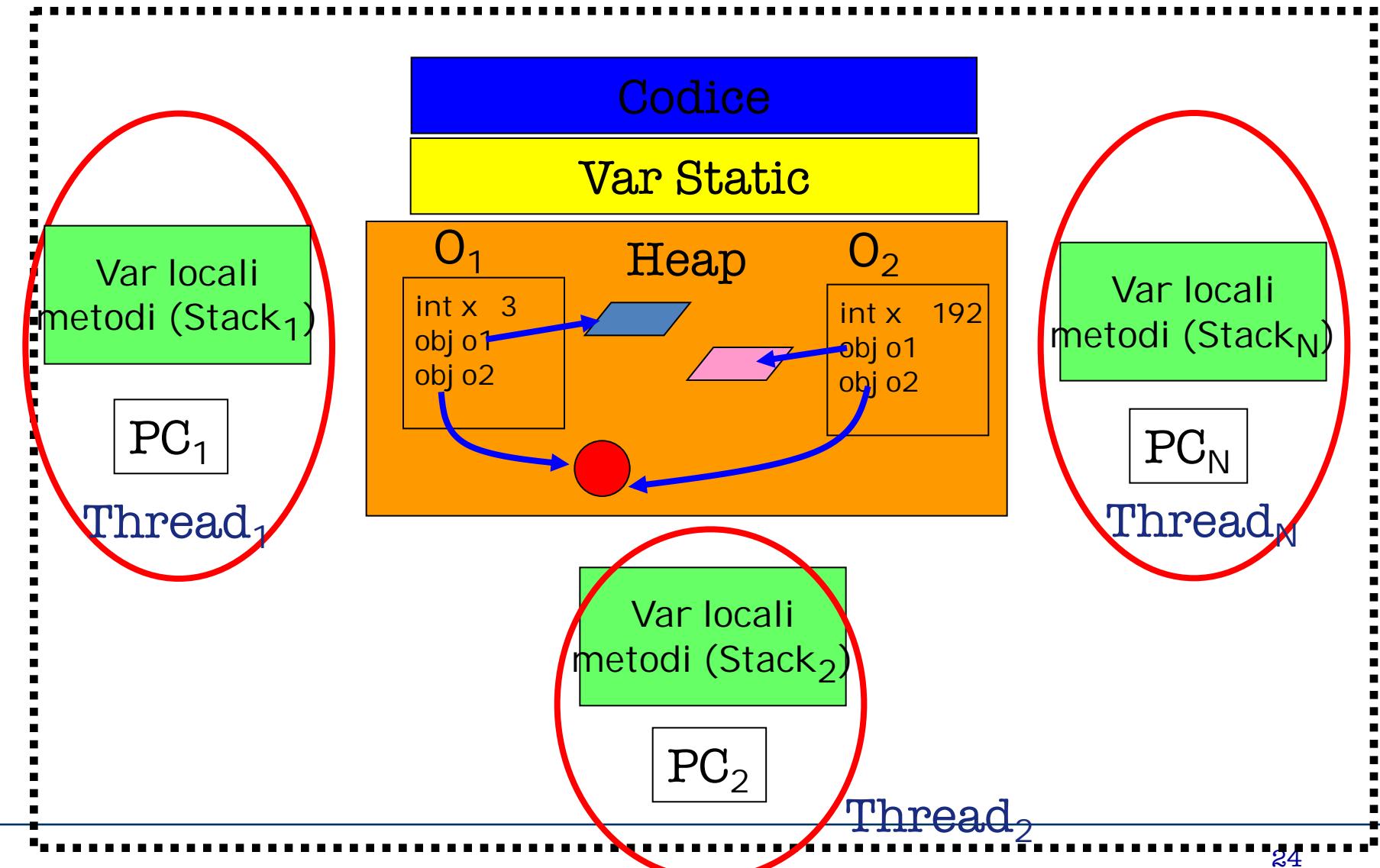
Mutua esclusione in Java:

metodi synchronized



Thread

Processo



-> O₁ e O₂ sono oggetti condivisi dai thread concorrenti

Mutua esclusione

In java è possibile denotare alcune sezioni di codice che operano su un oggetto in modo mutuamente esclusivo (cioè, sono **sezioni critiche**) tramite la parola chiave **synchronized**:

- **metodi synchronized**
- **[blocchi synchronized]**

-> Ogni parte di codice etichettata come **synchronized** viene eseguita sull'oggetto al quale viene riferita in modo **mutuamente esclusivo**, cioè da un thread alla volta.

Metodi synchronized

- **Mutua esclusione** tra i metodi di una classe

```
public class Contatore {  
    private int i=0;  
    public synchronized void incrementa()  
    { i++;  
        System.out.println("Il contatore è stato incrementato.  
        Nuovo valore: "+i+"!");  
  
    }  
    public synchronized void decrementa()  
    { i--;  
        System.out.println("Il contatore è stato decrementato.  
        Nuovo valore: "+i+"!");  
    }  
}
```

Metodi **synchronized**

Quando un metodo **synchronized** viene invocato da un thread T per operare su un oggetto O della classe, l'esecuzione del metodo avviene in **mutua esclusione**:

- ❑ se un altro thread sta eseguendo un metodo **synchronized** sullo stesso oggetto (l'oggetto O è **occupato**), il thread T viene **sospeso** ed inserito nella **coda (entry set)** associata ad O. Quando l'oggetto O tornerà libero, verrà riattivato il primo processo in coda → T uscirà dalla coda quando l'oggetto O sarà libero e non ci saranno più thread che lo precedono nell'entry set di O.

 - ❑ se nessun metodo **synchronized** sull'oggetto O è in esecuzione (l'oggetto è **libero**), il metodo viene eseguito (O viene occupato per tutta la durata della chiamata).
-

Esempio: accesso concorrente a un contatore

```
public class CompetingProc extends Thread
{ Contatore r; /* risorsa condivisa */
  int T; // incrementa se tipo=1; decrementa se tipo=-1

  public CompetingProc(Contatore R,  int tipo)
  {   this.r = R;
      this.T = tipo;
  }

  public void run()
  {   try{
      while(true)
      {   if (T>0)  r.incrementa();
          else if (T<0)  r.decrementa();
      }
      }catch(InterruptedException e){}

  }
}
```

```
public class Contatore {  
    private int C;  
  
    public Contatore(int i)  
    { this.C=i; }  
  
    public synchronized void incrementa()  
    { C++;  
        System.out.print("\n eseguito incremento: valore  
attuale del contatore: "+ C+ " ....\n");  
    }  
  
    public synchronized void decrementa()  
    { C--;  
        System.out.print("\n eseguito decremento: valore  
attuale del contatore: "+ C+ " ....\n");  
    }  
}
```

```
import java.util.*;  
  
public class Prova_mutex{ // test  
  
    public static void main(String args[]) {  
        final int NP=30;  
        Contatore C =new contatore(0);  
        CompetingProc []F=new CompetingProc[NP];  
        int i;  
        for(i=0;i<(NP/2);i++)  
            F[i]=new CompetingProc(C, 1); // incrementa  
        for(i=(NP/2);i<NP;i++)  
            F[i]=new CompetingProc(C, -1); // decrementa  
        for(i=0;i<NP;i++)  
            F[i].start();  
        for(i=0; i<NP; i++)  
            F[i].join();  
        System.out.print("\n main: tutti i figli sono  
terminati... Ciao!!\n");  
    } }  
  
Il metodo join() consente al thread di sincronizzarsi con la terminazione dei figli
```

Esercizio 1 (1/3)

Nella sala di un quartiere, un'associazione di volontariato ha organizzato un **evento benefico** a favore dei **profughi** delle zone di guerra. Allo scopo di sensibilizzare i cittadini, la sala ospita un'esposizione di foto scattate nei paesi in guerra.

L'evento prevede che ogni cittadino che vi partecipa contribuisca ad una **raccolta** di beni da inviare ai profughi.

La raccolta prevede che i beni da donare possano essere di tre tipi:

1. capi di **abbigliamento** (confezionati in sacchi)
2. **giocattoli**
3. **denaro**

Ogni **cittadino** che vuole contribuire alla raccolta con alcuni beni da donare:

1. **accede** alla sala, consegnando i propri contributi ai **volontari** presenti all'ingresso;
2. <permane nella sala per un tempo arbitrario, durante il quale visita l'esposizione fotografica>
3. **esce**.

Per motivi di sicurezza, gli accessi dei cittadini sono vincolati da una capacità massima **N**, che stabilisce il numero massimo di persone (esclusi i volontari) che contemporaneamente possono stare all'interno della sala.

Ogni cittadino, quindi, si reca all'ingresso e **richiede di entrare**:

- **nel caso vi sia posto**, entra occupando un posto e consegnando contestualmente i propri contributi ai volontari;
- **nel caso in cui la sala sia piena**, per evitare assembramenti il cittadino si allontana ed, eventualmente, dopo un pò di tempo ritorna alla sala per ritentare l'accesso.

Esercizio 1 (2/3)

Progettare un'applicazione java che regoli gli accessi alla sala, nella quale:

- ogni **cittadino** sia rappresentato da un **thread distinto**,
- la **sala** sia rappresentata da un **oggetto condiviso** da tutti i thread.

In particolare:

- ogni thread che tenta l'accesso alla sala e trova posto entra nella sala(aggiornandone coerentemente lo stato);
 - altrimenti attende un tempo arbitrario e riprova ad entrare.
 - Se dopo K tentativi il thread visitatore non trova posto, rinuncia alla visita e termina.
-

Esercizio 1 (3/3)

Il **thread iniziale** (**main**), una volta terminati tutti i **thread cittadini**, stampa:

- il numero totale di cittadini che hanno contribuito alla raccolta,
- il numero totale di sacchi di abbigliamento raccolti,
- il numero totale di giocattoli raccolti
- l'importo totale di denaro donato;

Successivamente il **thread iniziale** termina.

Impostazione

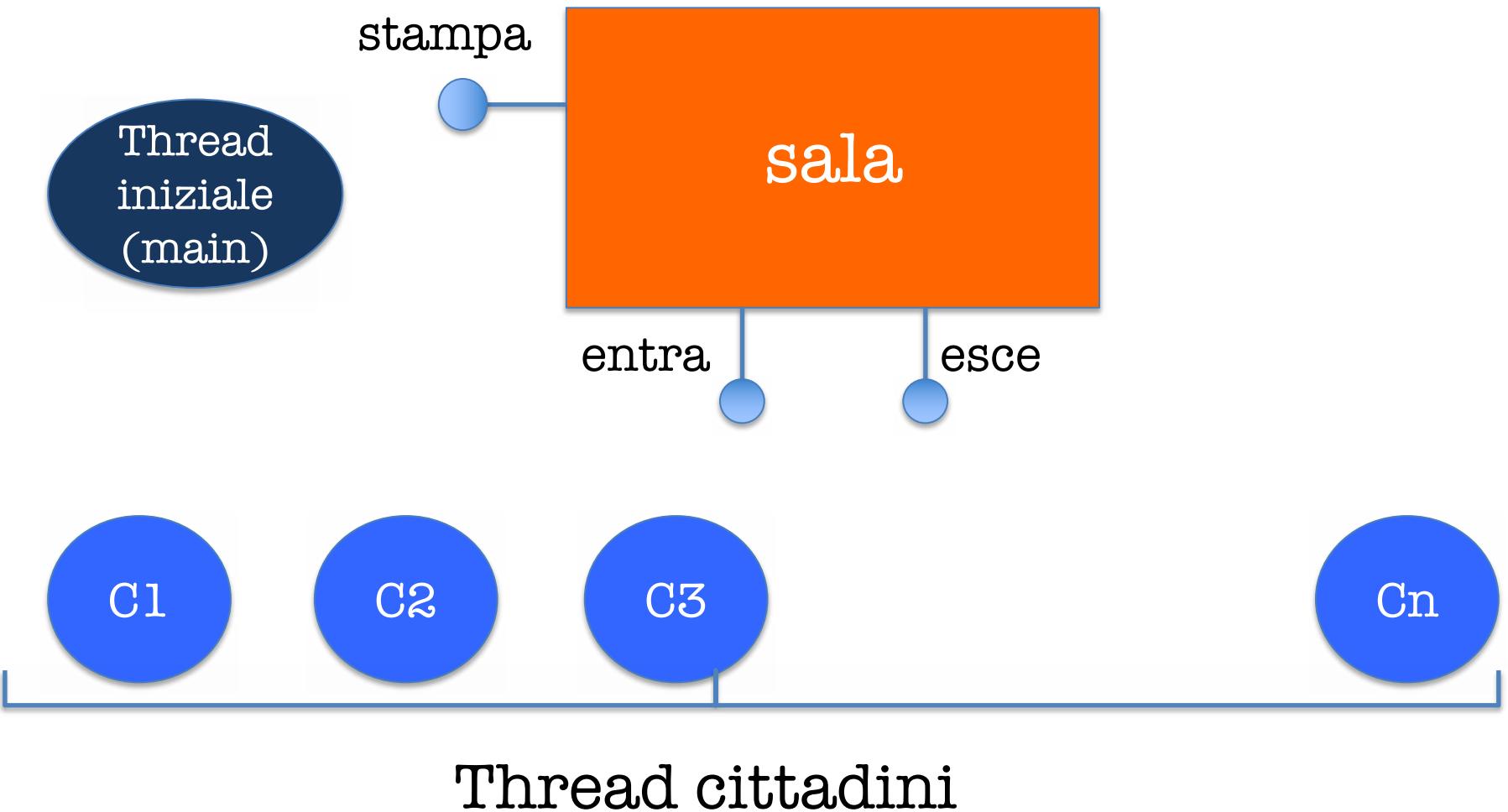
Supponiamo per semplicità che:

- Ogni cittadino, una volta entrato, permanga nella sala per un tempo arbitrario.
- i numeri di giocattoli, di sacchi di abbigliamento e il denaro donati da ogni cittadino siano definiti in modo casuale.

Classi da definire:

- **Sala**: è una risorsa condivisa acceduta in modo concorrente dai thread cittadini.
 - Quali variabili locali?
 - Quali metodi (necessità di sincronizzazione!)?
- **Cittadini**: thread
- **Main** (main)

Impostazione



Impostazione

Classi da definire:

- **Cittadino**: il generico thread concorrente che accede alla sala. Il suo comportamento è definito dal metodo run:

```
public class Visitatore extends Thread {  
    Sala S;  
    <costruttore, etc.>  
  
    public void run() {  
        int entrato;  
        ...  
        entrato = S.entra();  
        if (entrato){  
            <consegna e permanenza nella sala>  
            S.esci();}  
        else  
            ...  
            ...  
    }  
}
```

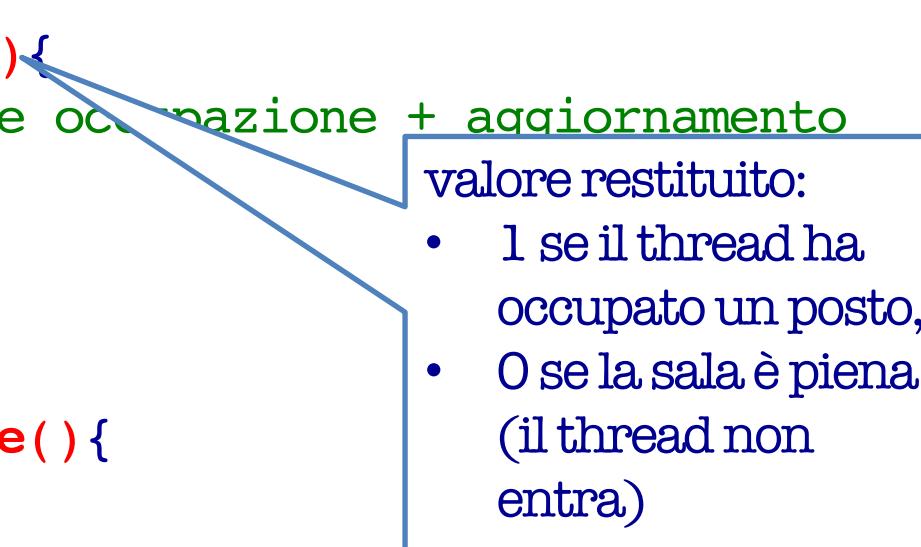
Verifica se c'è posto e lo occupa

libera il posto

Sala: è condivisa da thread concorrenti.

-> Usiamo i metodi **synchronized**.

```
public class Sala {  
    // var. locali: posti occupati, pacchi_abbigliamento,  
    giocattoli, donatori;  
  
    public synchronized int entra(){  
        <verifica posto + eventuale occupazione + raccolta>  
        return risultato;  
    }  
  
    public synchronized void esce(){  
        <liberazione posto>  
    }  
  
    public synchronized void stampa (){  
        <stampa risultati raccolta >  
    }  
}
```



+ aggiornamento
valore restituito:

- 1 se il thread ha occupato un posto,
- 0 se la sala è piena (il thread non entra)

Classe Main: contiene il metodo main

```
import java.util.Random;
public class Main{
    private final static int MAX_NUM= 20;

    public static void main(String[ ] args) {
        Random r = new Random(System.currentTimeMillis());
        int NC = r.nextInt(MAX_NUM);
        Cittadino[ ] V = new Cittadino[NC];
        Sala S= new Sala(...);
        <creazione NC Cittadini>
        <attivazione Cittadini>
        // il thread main deve aspettare la terminazione
        // dei cittadini prima di stampare i valori finali
        // -> uso del metodo join:
        for(i=0; i<NV; i++)
            V[i].join(); //attesa term. cittadino i-simo
        S.stampa(); //stampa dei valori finali
    }
}
```

Esercizio 2

Estendere l'esercizio 1 prevedendo che nell'edificio vi sia un bagno a disposizione di tutti i cittadini: sia quelli entrati nella sala, sia quelli fuori.

Il bagno ha capacità 1: una sola persona alla volta può usarlo.

Ogni cittadino entrato nella sala può utilizzare il bagno continuando ad occupare il posto nella sala.

Per accedere al bagno, ogni cittadino:

- se trova il bagno occupato, attende fino a quando non torna libero e non ci sono più persone in attesa davanti a lui;
 - quando il bagno è libero lo occupa per un tempo arbitrario.
-

Nona Esercitazione

Thread e memoria condivisa
Sincronizzazione tramite semafori



Semafori in Java

Dalla versione 5.0, è disponibile la classe Semaphore:

```
import java.util.concurrent.Semaphore;
```

tramite la quale si possono creare semafori, sui quali è possibile operare tramite i metodi:

- **acquire();** // implementazione di **p()**
- **release();** // implementazione di **v()**

Uso di oggetti **Semaphore**:

Inizializzazione ad un valore K dato:

```
Semaphore s=new Semaphore(k);
```

Operazioni: stessa semantica di p e v

```
s.acquire(); // esecuzione di p() su s
```

```
s.release(); // esecuzione di v() su s
```

Esempio sui Semafori

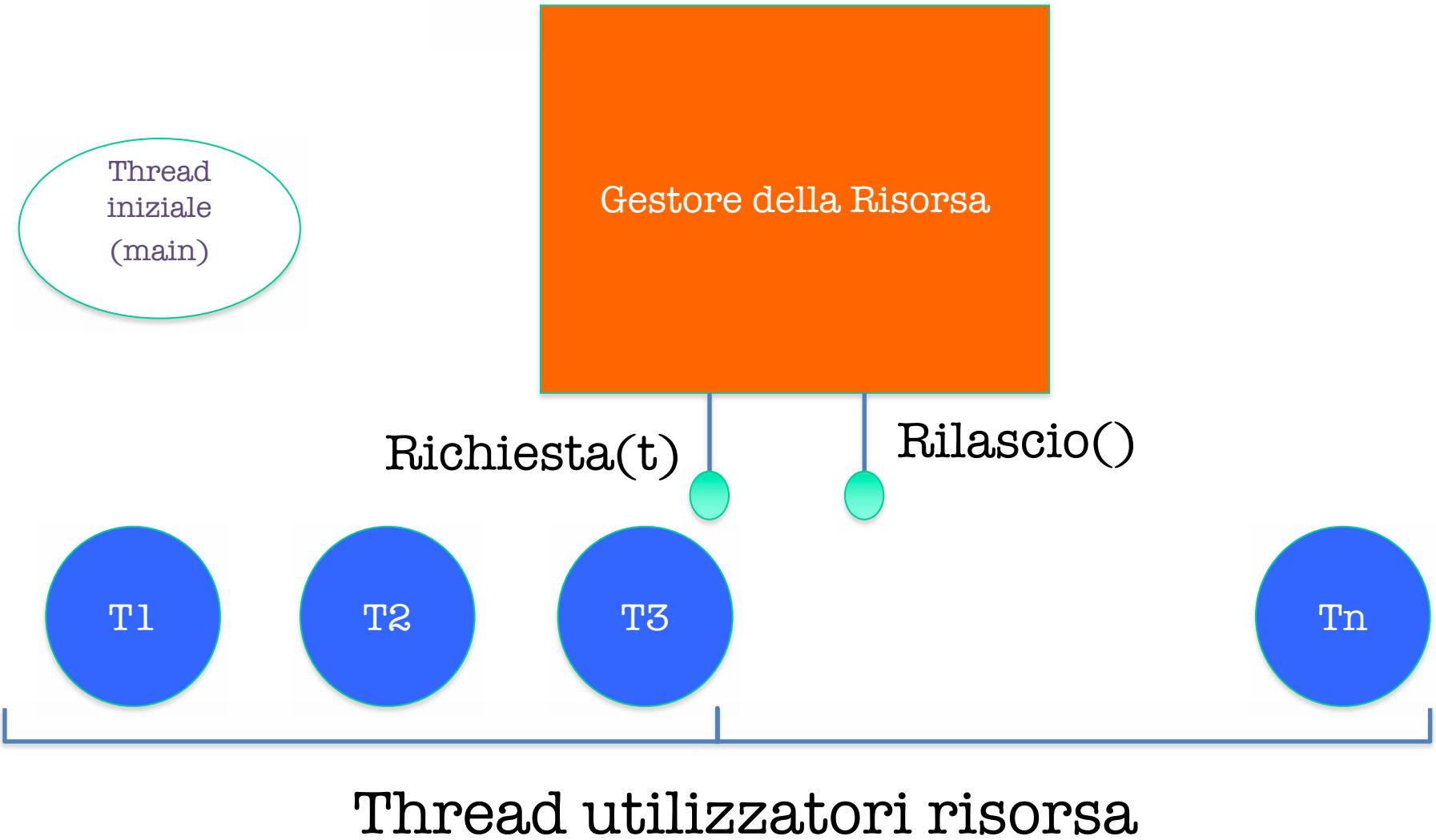
Allocazione di una risorsa con politica prioritaria (SJF)

Traccia (1/2)

Si realizzi una applicazione Java che risolva il problema dell'allocazione di una risorsa secondo la politica “**Shortest Job First**”, ovvero:

- **Una sola risorsa condivisa** da più thread
 - Ogni thread utilizza la risorsa:
 - In modo **mutuamente esclusivo**
 - In modo **ciclico**
 - Ogni volta, **per una quantità di tempo arbitraria** (stabilita a run-time e dichiarata al momento della richiesta).
 - **Politica di allocazione della risorsa:**
 - **SJF**: La precedenza va al thread che intende utilizzarla per il minor tempo.
-

Impostazione



Impostazione

- Quali classi ?
 - **ThreadP**: thread utilizzatori della risorsa; struttura ciclica e determinazione casuale del tempo di utilizzo
 - **Gestore**: mantiene lo stato della risorsa e implementa la politica di allocazione basata su priorità:
 - Richiesta(t) [t è il tempo di utilizzo]: **sospensiva** se
 - la risorsa è occupata,
 - oppure se c'è almeno un processo **più prioritario** (cioè che richiede un tempo minore di t) in attesa
 - Rilascio(): **rilascio** della risorsa ed eventuale **risveglio** del processo più prioritario in attesa (quello che richiede il minimo t tra tutti i sospesi).
 - **SJF**: classe di test (contiene il main())

Soluzione: classe ThreadP

```
import java.util.Random;

public class ThreadP extends Thread{
    Gestore g;
    Random r;
    int maxt;

    public ThreadP(Gestore G, Random R, int MaxT)
    {
        this.r=R;
        this.g=G;
        this.maxt=MaxT;
    }
}
```

```
public void run(){
    int i, tau;  long t;
    try{
        this.sleep(r.nextInt(5)*1000);
        tau=r.nextInt(maxt);
        for(i=0; i<15; i++)  {
            g.richiesta(tau);
            this.sleep(tau);
            System.out.print("\n["+i+"] Thread:"+getName()
                +" e ho usato la CPU per "+tau+"ms...\n");
            g.rilascio();
            tau=r.nextInt(maxt); // calcolo nuovo tau
        }
    }catch(InterruptedException e){}
} //chiude run
}
```

Uso della risorsa.
UN SOLO THREAD
ALLA VOLTA!

Impostazione del gestore

Due cause di sospensione:

1. Accessi al Gestore della risorsa mutamente esclusivi: 1 alla volta! => definisco un semaforo di mutua esclusione

```
semaphore mutex = new Semaphore(1);
```

2. La risorsa è occupata, oppure c'è almeno un thread più prioritario in attesa:

Quando la risorsa viene liberata deve essere svegliato il processo più prioritario => creiamo un semaforo per ogni livello di priorità:

```
semaphore [ ]codaproc; //1 per ogni liv. Priorità
```

Poichè ogni semaforo serve per sospendere processi la cui richiesta non può essere soddisfatta, ogni elemento di codaproc va inizializzato a 0 (semaforo “rosso”).

Necessità di individuare quanti siano i processi in attesa e la loro priorità:

```
int [ ]sospesi; //contatori thread sospesi
```

Classe Gestore

```
public class Gestore {  
    int n;                                // massimo tempo di uso della risorsa  
    boolean libero;  
    Semaphore mutex;           //semaforo x la mutua esclusione  
    Semaphore []codaproc; //1 coda per ogni liv. Prio (tau)  
    int []sospesi;                //contatore thread sospesi  
  
    public  Gestore(int MaxTime) {  
        int i; this.n=MaxTime;  
        mutex = new Semaphore(1);  
        sospesi = new int[n];  
        codaproc = new Semaphore[n];  
        libero = true;  
        for(i=0; i<n; i++) {  
            codaproc[i]=new Semaphore(0); //semafori "condizione"  
            sospesi[i]=0;  
        } }                                // continua...
```

...classe Gestore

```
/*richiesta per tau ms*/
public void richiesta(int tau){
    int i=0;
    try{
        mutex.acquire();
        while(più_prio(tau) || libero==false){
            sospesi[tau]++;
            mutex.release();
            codaproc[tau].acquire();
            mutex.acquire();
            sospesi[tau]--;
        }
        libero = false;
        mutex.release();
    }catch(InterruptedException e){}
}
```

verifico che ci sia un processo più prioritario in attesa

...classe Gestore

// .. Continua

```
public void rilascio() {
    int da_svegliare, i;
    try{
        mutex.acquire();
        libero=true;
        da_svegliare = min_sosp();
        if (da_svegliare>=0)
            codaproc[da_svegliare].release();
        mutex.release();
    }catch(InterruptedException e){}
}
```

Sveglio il processo più prioritario in attesa.

...classe Gestore (metodi utili)

```
private boolean piu_prio(int tau){  
    int i=0;  
    boolean risposta=false;  
    for(i=0; i<tau; i++)  
        if (sospesi[i]!=0)  
            return true;  
    return risposta;
```

```
}
```

c'è qualcuno più prioritario del thread che userà la risorsa per tau secondi?

```
private int min_sosp(){  
    int i=0, ris=-1;  
    for(i=0; i<n; i++)  
        if (sospesi[i]!=0)  
            return i;  
    return ris;
```

```
}
```

Chi è il processo più prioritario (con minor tau) sospeso in coda?

Soluzione: classe sjf

```
import java.util.*;
import java.util.Random;

public class sjf{
    public static void main(String args[]) {
        final int NT=10;//thread
        final int MAXT=500; // quanto di tempo massimo
        int i;
        Random r=new Random(System.currentTimeMillis());
        threadP []TP=new threadP[NT];
        gestore G=new gestore(MAXT);
        for (i=0; i<NT; i++)
            TP[i]=new threadP(G, r, MAXT);
        for (i=0;i<NT; i++)
            TP[i].start();
    }
}
```

Commento finale

- Nell'esempio abbiamo usato un semaforo per ogni livello di priorità; ogni semaforo è quindi usato per accodare processi con la medesima priorità ed è gestito in modo tale che ogni **p** (`acquire()`) risulti **sospensiva**.
- semafori utilizzati in questo modo vengono detti «**semafori privati**» perchè ogni semaforo serve per sospendere i processi di un certo tipo (es: il semaforo `codapro[k]` per i processi che hanno priorità= k); Pertanto si dice che il semaforo è «privato» per quella categoria di processi.

Schema tipico di uso dei semafori privati:

```
//ACQUISIZIONE RISORSA:
```

```
mutex.acquire();
while(<condizione di sospensione>){
    sospesi_k++;
    mutex.release();
    s_k.acquire();
    mutex.acquire();
    sospesi_k--;
}
<aggiornamento stato risorsa condivisa>
mutex.release();
```

```
//RIATTIVAZIONE Thread sospesi:
```

```
mutex.acquire();
...
<individuazione del tipo di processo k da
riattivare>
if ( <c'è almeno un processo nella coda di s_k > )
    s_k.release();
...
mutex.release();
```

Esercizio 1 - La gita in battello

In un'isola turistica vengono organizzate delle gite in battello per i turisti che vogliono ammirare la costa circumnavigando l'isola.

Ogni gita si svolge utilizzando un battello messo a disposizione dall'ente di promozione turistica della zona.

Il battello ha una capienza data **C_{MAX}** che esprime il numero massimo di turisti che possono essere a bordo del battello.

Ogni gita parte e termina dallo stesso molo; affinchè una gita si possa svolgere è necessario che il battello sia pieno.

Sul battello è costantemente presente un comandante che decide quando partire e quando terminare la gita.

Si assume che il **comandante** del battello e **ogni turista** siano rappresentati da **thread concorrenti**, e che il **battello** sia la risorsa condivisa.

Comportamento di ogni **turista**:

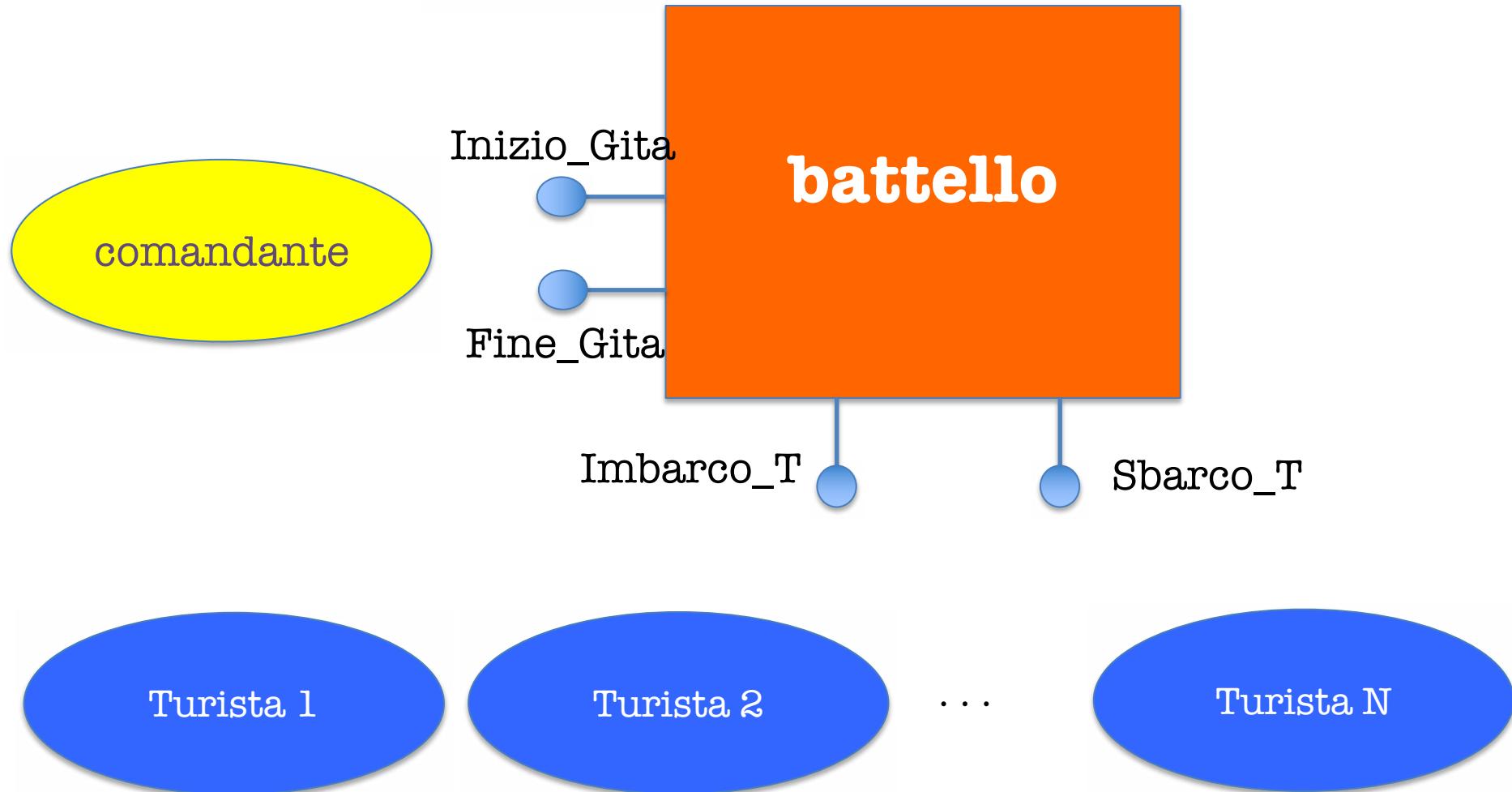
- **si imbarca sul battello**
- <partecipa alla gita>
- **sbarca dal battello**

Comportamento del **comandante**:

ciclicamente:

- **dà inizio alla gita**
- <conduce il battello nella circumnavigazione dell'isola>
- **termina la gita**

Impostazione



Impostazione – quali classi?

- **Turista**: thread che rappresenta il singolo partecipante alla gita.
- **Comandante**: thread che rappresenta il comandante del battello.
- **Battello**: risorsa condivisa; contiene lo stato del Battello e implementa i metodi:
 - **Imbarco_T()**: **sospensivo** se il battello è pieno, oppure se c'è una visita in corso.
 - **Sbarco_T()**: **sospensivo** se la gita non è terminata.
 - **Inizio_gita()**: **sospensivo** se il battello non ha ancora imbarcato il numero massimo di turisti.
 - **Fine_gita()**: il battello termina il giro; viene consentito ai turisti di sbarcare;
- Ogni metodo deve essere eseguito in mutua esclusione.
- **Gite_in_battello**: definisce il metodo main, che crea le istanze di tutte le altre classi e attiva tutti i thread.

Classe Battello

- E' la risorsa condivisa tra i thread: definisce lo stato del battello e implementa la sincronizzazione tra thread.
- Quanti semafori?
 - **Mutua esclusione:** per rendere mutuamente esclusiva l'esecuzione di ogni metodo sulla risorsa Battello
 - Sospensione Turisti in imbarco: **STI**
 - Sospensione Turisti in sbarco: **STS**
 - Sospensione Comandante: **SC**

Suggerimento: gestire STI, STS e SC come semafori privati.

Esercizio 2 - La gita guidata

Estendere l'es.1 prevedendo che ogni gita sia guidata.

A questo scopo nell'isola sono disponibili M guide turistiche volontarie, che all'occorrenza possono accompagnare i turisti nella gita.

In particolare si assuma che **per ogni gita** siano necessarie **2 guide** turistiche: una illustrerà la gita al gruppo di turisti che si trovano sul ponte del battello, l'altra a quelli che si trovano nella parte coperta della barca (sottocoperta) .

Il battello ha una capienza data **C_{MAX}** che esprime il numero massimo di turisti che possono essere a bordo del battello.

Ogni gita parte e termina dallo stesso molo; affinchè una gita si possa svolgere è necessario che entrambe le seguenti condizioni siano vere:

- **vi siano 2 guide a bordo del battello;**
- **il battello sia pieno.**

Sul battello è costantemente presente un comandante che decide quando partire e quando terminare la gita.

Sia assunta che il comandante del battello, ogni turista e ogni guida siano rappresentati da thread concorrenti, e che il battello sia la risorsa condivisa.

Comportamento di ogni turista:

- **si imbarca sul battello**
- <partecipa alla gita ascoltando le spiegazioni delle guide>
- **sbarca dal battello**

Comportamento di ogni guida:

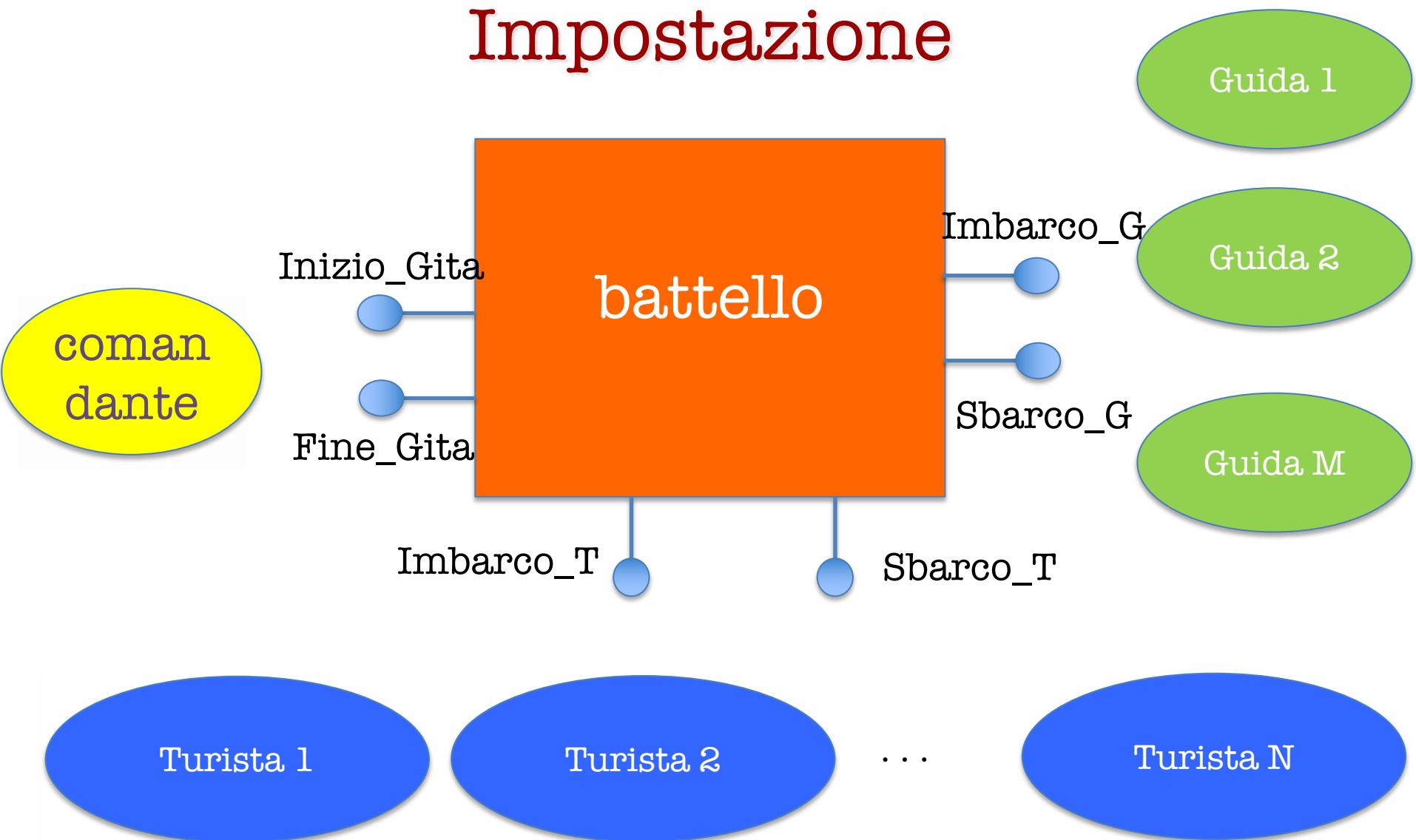
- **si imbarca sul battello**
- <illustra la gita al proprio gruppo di turisti>
- **sbarca dal battello**

Comportamento di ogni comandante:

ciclicamente:

- **dà inizio alla gita**
 - <conduce il battello nella circumnavigazione dell'isola>
 - **termina la gita**
-

Impostazione



Impostazione – quali classi?

Rispetto all'esercizio 1:

- è necessario aggiungere una nuova **classe Guida**: thread che rappresenta la singola guida
- è necessario **modificare la classe Battello**:
 - aggiungendo i metodi usati dalle guide
 - aggiungendo i semafori necessari alla sincronizzazione delle guide (in quali situazioni si può sospendere una guida?)
 - modificando alcuni degli altri metodi per gestire correttamente la sincronizzazione. (es: il comandante non può iniziare la gita se non ci sono entrambe le guide necessarie).

Decima Esercitazione

Accesso a risorse condivise tramite
Monitor in Java

Agenda

Esempio

L'album delle figurine: gestione di una risorsa condivisa da più thread, con politica prioritaria

Esercizio 1 - CAF

Esercizio 2 - CAF con priorità

Esempio - La collezione di figurine (1/3)

Una casa editrice vuole realizzare un **sito web** dedicato ai collezionisti di figurine dell'album **“Campionato di calcio 2021-2022”**.

L'album è composto da **N=100 diverse figurine**, ognuna individuata univocamente da un id intero [0, 99]; tra di esse:

- **30** sono classificate come **figurine rare** (id da 0 a 29);
- le rimanenti **70** sono classificate come **figurine normali** (id da 30 a 99).

Il sito offre un servizio che permette ad ogni utente collezionista di effettuare **scambi di figurine**.

A questo scopo il sistema gestisce un **deposito di figurine**, nel quale, **per ogni diversa figurina vi può essere più di un esemplare**.

Esempio - La collezione di figurine (2/3)

Il meccanismo di scambio, è regolamentato come segue:

- Si può scambiare solo **una figurina alla volta**, effettuando una **richiesta di scambio** con le seguenti regole:
 - ogni **utente U che desidera una figurina A può ottenerla, se a sua volta offre un'altra figurina B**;
 - in seguito a una richiesta di scambio, il **sistema aggiunge la figurina B all'insieme delle figurine disponibili e successivamente verifica se esiste almeno una figurina A disponibile:**
 - se **A è disponibile**, essa viene assegnata all'utente U, che può così continuare la propria attività;
 - se **A non è disponibile**, l'utente U viene messo in attesa.

Esempio - La collezione di figurine (3/3)

Si progetti la politica di gestione del servizio di scambio che tenga conto delle specifiche date e che inoltre soddisfi il seguente vincolo:

le richieste di **utenti che offrono figurine rare abbiano la precedenza sulle richieste di utenti che offrono figurine normali.**

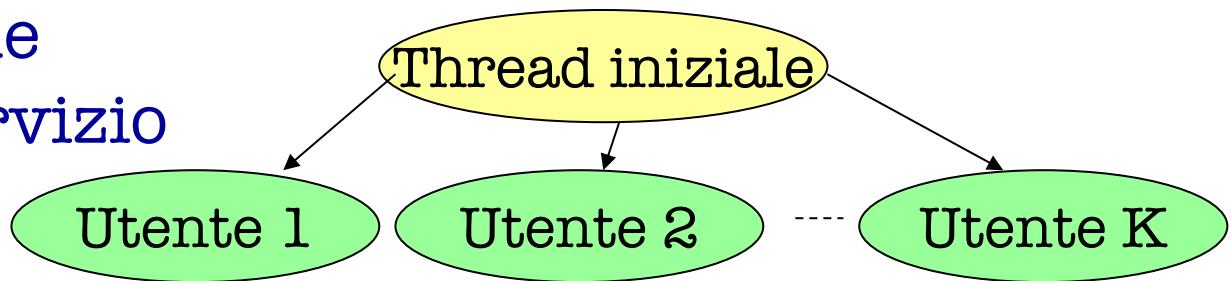
Ad esempio:

1. Il thread TA chiede 7 offrendo 3[RARA]; 7 non disponibile → **TA attende**
2. Il thread TB chiede 7 offrendo 50[NORM]; 7 non disponibile → **TB attende**
3. 7 diventa disponibile => deve essere **attivato TA** (perchè offre una rara, quindi è più prioritario).

Impostazione

Quali thread?

- il thread iniziale
- K utenti del servizio



Qual è la risorsa comune?

- ! Deposito delle Figurine
- ! associamo al Deposito un "**monitor**", che controlla gli accessi in base alla specifica politica di accesso. La sincronizzazione viene realizzata mediante **variabili condizione**.

Thread Collezionista

```
public class Collezionista extends Thread{  
    private Monitor M;  
    private int offerta, richiesta, N;  
  
    public Collezionista(monitor m, int N) {  
        this.M=m;  
        this.N=N;  
    }  
    public void run() {  
        try { while (true) {  
            <definizione di offerta e richiesta>  
            M.scambio(offerta, richiesta); //entry call  
            Thread.sleep(...);  
        } } catch (InterruptedException e) { }  
    }  
}
```

riferimento al monitor

numero di figurine diverse. Se ci atteniamo alle specifiche, N=100.

Monitor – Deposito figurine

Stato del Deposito:

Figurine disponibili: vettore di $N=100$ interi (uno per ogni figurina della collezione)

```
private int[] FIGURINE = new int[N];
```

Dove: `FIGURINE[i]` è il numero di esemplari disponibili della figurina i .

(Hp: inizialmente 1 per ogni figurina)

Convenzione adottata:

Se $i < 30$, si tratta di una **figurina rara;**

Se $i \geq 30$, si tratta di una **figurina comune.**

Monitor – Deposito figurine

Lock per la mutua esclusione:

```
private Lock lock = new ReentrantLock();
```

Condition. Per la sospensione dei thread in attesa di una figurina, definiamo 2 condition (una per ogni livello di priorità):

```
private Condition rare= ....;
//thread sospesi che hanno offerto figurine rare
private Condition normali=....;
// thread sospesi che hanno offerto figurine
normali
```

Contatori dei thread sospesi in ogni coda:

```
private int[] sospRare = new int[N];
private int[] sospNormali = new int[N];
//devo sapere chi è sospeso in attesa di quella
specifica figurina dopo aver consegnato una
figurina rara o una normale
```

Monitor – Deposito figurine

```
public class Monitor {  
    private final int N=100; //numero totale di figurine  
    private final int maxrare=30;  
    private int[] FIGURINE; //figurine disponibili  
    private Lock lock = new ReentrantLock();  
    private Condition rare = lock.newCondition();  
    private Condition normali = lock.newCondition();  
    private int[] sospRare;  
    private int[] sospNormali;  
  
    public Monitor() {...} //Costruttore  
    public void scambio(int off,int rich)//metodo entry:  
        throws InterruptedException {...}  
}
```

politica di
sincronizzazione

Soluzione

```
import java.util.Random;
public class Collezionista extends Thread{
    private Monitor M;
    private int offerta, richiesta, max;

    public Collezionista(Monitor m, int NF, String name){
        this.M=m; this.max=NF;
        this.setName(name);
    }

    public void run(){
        int op, cc, somma;
        try { while (true)
            {
                offerta= r.nextInt(max);
                do { richiesta= r.nextInt(max);
                    }while(richiesta==offerta);
                M.scambio(offerta, richiesta);
                Thread.sleep(250);
            }
        } catch (InterruptedException e) { }
    }
}
```

Soluzione: monitor

```
import java.util.concurrent.locks.*;  
  
public class Monitor{ //Dati:  
    private int N; //numero totale di figurine  
    private final int maxrare;  
    private int[] FIGURINE= new int[N];; //figurine disponibili  
  
    private Lock lock= new ReentrantLock();  
    private Condition rare= lock.newCondition();  
    private Condition normali= lock.newCondition();  
    private int[] sospRare= new int[N];  
    private int[] sospNormali= new int[N];  
  
    public Monitor(int N ) {  
        int i;  
        for(i=0; i<N; i++) {  
            FIGURINE[i]=1;//valore arbitrario  
            sospRare[i]=0;  
            sospNormali[i]=0;  
        }  
        maxrare=N/100*30; }  
}
```

```
//metodi "entry":  
public void scambio(int off, int rich) throws InterruptedException {  
    lock.lock();  
    try{        FIGURINE[off]++;  
        if (sospRare[off]>0)  
            rare.signalAll();  
        else if (sospNormali[off]>0)  
            normali.signalAll();  
        if (off < maxrare) //ha offerto una figurina rara  
            while (FIGURINE[rich]==0){  
                sospRare[rich]++;  
                rare.await();  
                sospRare[rich]--; }  
        else //ha offerto una normale  
            while (FIGURINE[rich]==0 || sospRare[rich]>0){  
                sospNormali[rich]++;  
                normali.await();  
                sospNormali[rich]--; }  
        FIGURINE[rich]--; // tolgo la figurina scambiata  
    } finally{ lock.unlock();}  
    return;  
}
```

perchè
signalAll?

Commenti finali

- La soluzione prevede solo 2 condition (rare, normali), ma le condizioni di sincronizzazione possibili sono $2 * 100$: per ogni categoria (rare/normali), ogni thread si sospende in attesa di una particolare figurina.
- Per evitare le signalAll (poco efficienti) si potrebbero definire $2 * 100$ condition:

```
private Condition []rare=new Condition[N];  
//code thread che hanno offerto rare  
  
private Condition []normali=new Condition[N];  
//code thread hanno offerto normali
```

Esercizio 1

Si consideri un Centro di Assistenza Fiscale (CAF) che offre servizi di consulenza fiscale ai propri utenti.

Gli utenti possono accedere al CAF senza prenotazione, purché rispettino alcuni vincoli.

I servizi del CAF vengono forniti da N consulenti, ognuno dei quali ciclicamente fornisce consulenza a un diverso utente. Si assume che ogni consulente possa servire un utente alla volta.

Ogni consulente, nel caso in cui non stia servendo nessun utente, può uscire dal CAF per una pausa di durata arbitraria, al termine della quale rientra in ufficio.

Il CAF ha una capienza limitata fissata a MAX , che esprime il massimo numero di persone (utenti e consulenti) che contemporaneamente possono stare all'interno del centro.

Un utente U può entrare nel CAF se c'è almeno un consulente libero e se il centro non è pieno; contestualmente all'ingresso, U verrà affidato a un consulente C fino al momento della uscita di U. A questo proposito si tenga presente che tutti i consulenti sono equivalenti, pertanto ad ogni utente verrà assegnato uno qualunque tra i consulenti disponibili.

Un Consulente può **entrare** nel CAF solo se il centro non è pieno. Un consulente può **uscire** dal CAF solo se non ha un utente da servire.

Realizzare un'applicazione concorrente in Java basata sul monitor nella quale Consulenti e Utenti siano rappresentati da thread concorrenti.

La sincronizzazione tra i thread dovrà tenere conto di tutti i vincoli dati.

Impostazione

Quali thread?

- thread iniziale
- Utenti del CAF
- Consulenti

Quale risorsa comune?

- il CAF

Associamo al CAF un "**monitor**", che controlla gli accessi in base alla politica di accesso.

La sincronizzazione viene realizzata mediante **variabili condizione**.

Struttura dei thread

```
public class Utente extends Thread{  
    private Monitor M;  
  
    public Utente(...){ // costruttore...  
    }  
  
    public void run(){  
        ...  
        M.entraU();  
        <riceve consulenza>  
        M.esceU();  
    }  
}
```

Struttura dei thread

```
public class Consulente extends Thread{  
    private Monitor M;  
    public Consulente(...){ // costruttore..  
    }  
  
    public void run(){  
        while (...) {  
            M.entraC();  
            <permanenza nel CAF, durante la quale  
            può fornire consulenze>  
            M.esceC();  
        }  
    }  
}
```

Monitor: gestisce il CAF

Variabili di stato:

Consulenti: quanti commessi si trovano all'interno del CAF; quali di essi sono disponibili e quali assegnati a un utente.

Posti: posti occupati nel CAF (**da utenti e consulenti**)

Politica di Sincronizzazione

Un thread **Utente** si sospende in ingresso:

- se non c'è posto nel CAF
- se non c'è un Consulente libero

Un thread **Consulente** si può sospendere:

- in Ingresso: se non c'è posto nel CAF
- in Uscita: se sta servendo un Utente

si può sospendere?

→ quante condition?

Esercizio 2

Si estenda l'esercizio 1, prevedendo che ogni utente possa appartenere a una delle 2 categorie seguenti:

- **UI**: Utente che chiede consulenza sull'ISEE;
- **UR**: Utente che chiede consulenza sulla dichiarazione dei Redditi;

La politica di accesso al CAF dovrà rispettare tutti i vincoli dati ed inoltre **nell'ingresso al CAF**:

- deve **privilegiare i Consulenti rispetto agli Utenti**;
- tra gli **Utenti**, deve dare la **precedenza agli utenti UR**.

Esercitazione 11

Gruppo [A-K]

Monitor Avanzato

Agenda

Esempio – Gestione di un ponte

Esercizio 1 – La mostra

Gestione di accessi/uscite da una sala di esposizione attraverso un corridoio

Esercizio 2 – La mostra con gruppi di numerosità variabile.

Esempio

Gestione di un ponte

Esempio

Si consideri un piccolo ponte che collega le due rive (Nord e Sud) di un fiume.

Al ponte possono accedere due tipi di veicoli: auto e moto.

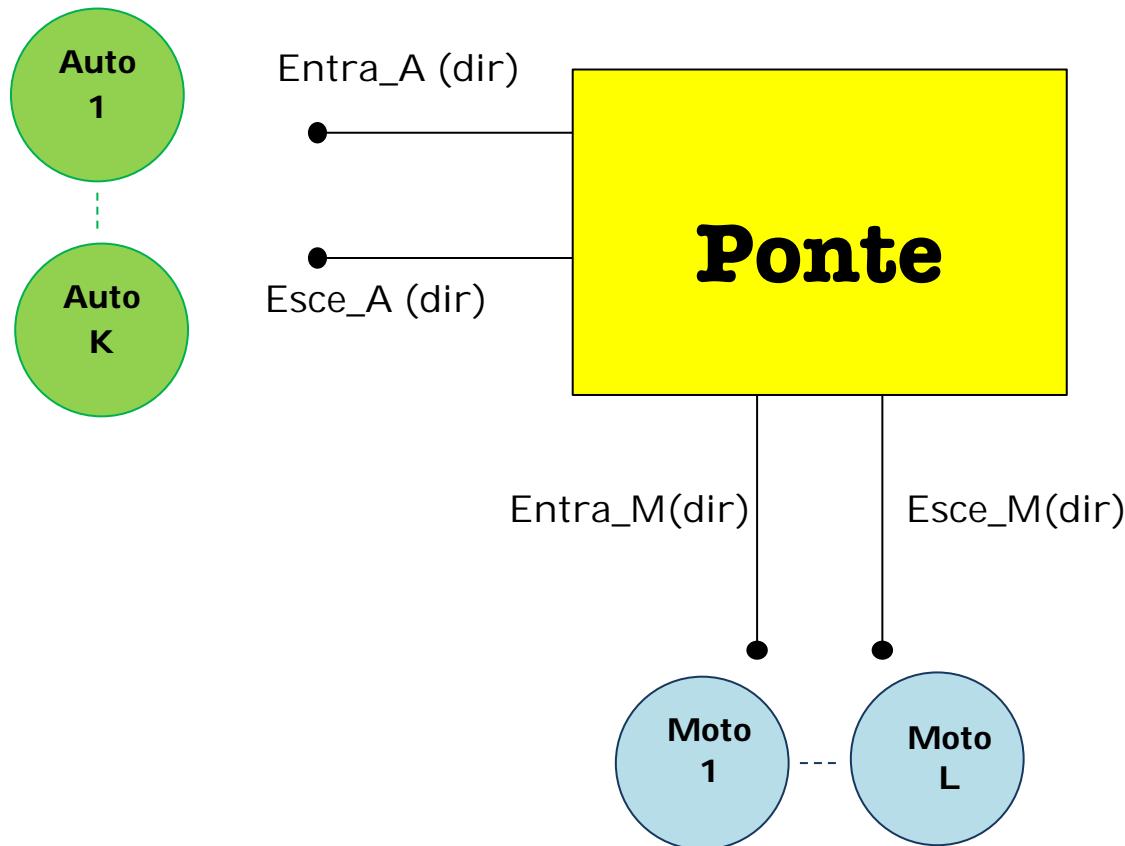
Il ponte ha una capacità massima MAX che esprime il numero massimo di motoveicoli che possono transitare contemporaneamente su di esso: a questo proposito si assuma che un'automobile valga come 4 motoveicoli.

Il ponte è talmente stretto che il transito di un'auto in una particolare direzione d impedisce l'accesso al ponte di qualunque altro veicolo (auto o moto) in direzione opposta a d.

Realizzare una politica di sincronizzazione delle entrate e delle uscite dal ponte che tenga conto delle specifiche date e che, nell'accesso al ponte, dia la **precedenza alle moto**, rispetto alle auto.

Impostazione

- Quali sono i thread? → **auto e moto**
- Qual è la risorsa condivisa? → **il ponte**



Sincronizzazione

Possibile sospensione di auto e moto nell'accesso al ponte.

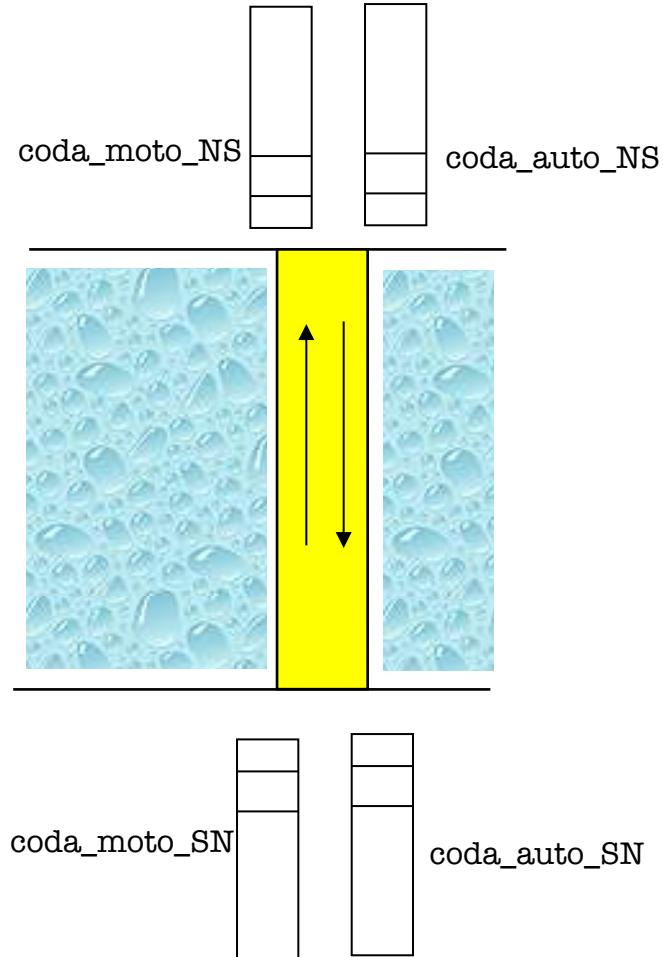
- Una Moto si sospende in ingresso:
 - se il ponte è pieno
 - se c'è almeno un'auto sul ponte in direzione opposta
- Un'auto si sospende in ingresso:
 - se il ponte è pieno
 - se c'è almeno un'auto o una moto sul ponte in direzione opposta
 - se c'è almeno una moto in attesa (in qualunque direzione)

Sincronizzazione

Quante/quali condition?

La sospensione di auto e moto dipende anche dalla direzione di accesso:

- 2 code di accesso sulla riva Nord (veicoli N->S)
 - **coda_auto_NS**
 - **coda_moto_NS**
- 2 code di accesso sulla riva Sud (veicoli S->N)
 - **coda_auto_SN**
 - **coda_moto_SN**



Soluzione: thread Auto

```
public class Auto extends Thread {  
    private Monitor M;  
    private int dir;  
    private Random r;  
  
    public Auto(Monitor M, int D, Random R, int i) {  
        this.M = M;  
        this.dir=D;  
        this.r=R;  
    }  
  
    public void run() {  
        try { sleep(r.nextInt(10*1000));  
            M.entraAuto(dir);  
            sleep(r.nextInt(10*1000));  
            M.esceAuto(dir);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Soluzione: thread Moto

```
public class Moto extends Thread {  
    private Monitor M;  
    private int dir;  
    private Random r;  
  
    public Moto(Monitor M, int D, Random R, int i) {  
        this.M = M;  
        this.dir=D;  
        this.r=R;  
    }  
  
    public void run() {  
        try { sleep(r.nextInt(10*1000));  
            M.entreMoto(dir);  
            sleep(r.nextInt(10*1000));  
            M.esceMoto(dir);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Soluzione: Monitor

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Monitor {
    //costanti di direzione:
    private final int NS=0;
    private final int SN=1;
    private int MAX;
    private Lock lock = new ReentrantLock();
    private int []autoIN=new int[2]; // auto sul ponte, per ogni dir
    private int []motoIN=new int[2]; // moto sul ponte, per ogni dir
    private int totIN; //numero totale di moto equivalenti sul ponte
    // Condition e contatori dei sospesi:
    private Condition [] codaAuto = new Condition[2]; //1 coda x dir
    private Condition [] codaMoto = new Condition[2]; //1 coda x dir
    private int []sospAuto=new int[2];
    private int []sospMoto=new int[2];
```

Soluzione: Monitor

```
//Costruttore:  
  
public Monitor(int N) {  
    this.MAX= N;  
    this.totIN=0;  
    for (int i=0; i<2;i++)  
    {        autoIN[i]=0;  
        motoIN[i]=0;  
        codaAuto[i]=lock.newCondition();  
        codaMoto[i]=lock.newCondition();  
        sospAuto[i]=0;  
        sospMoto[i]=0;  
    }  
  
}  
private int altradir(int d) // ritorna la direz. opposta a d  
{    if (d==NS)  
        return SN;  
    else  
        return NS;  
}
```

Soluzione: Monitor

```
public void entraMoto(int d) throws InterruptedException
{
    lock.lock();
    while (totIN==MAX || (autoIN[altradir(d)]) >0 ) {
        sospMoto[d]++;
        codaMoto[d].await();
        sospMoto[d]--;
    }
    motoIN[d]++;
    totIN++;
    lock.unlock();
}
```

Soluzione: Monitor

```
public void entraAuto(int d) throws InterruptedException
{
    lock.lock();
    while (totIN+4>MAX || autoIN[altradir(d)] >0 || motoIN[altradir(d)]>0 || sospMoto[NS]>0 || sospMoto[SN]>0) {
        sospAuto[d]++;
        codaAuto[d].await();
        sospAuto[d]--;
    }
    autoIN[d]++;
    totIN+=4;
    lock.unlock();
}
```

Soluzione: Monitor

```
public void esceMoto(int d) {  
    lock.lock();  
    motoIN[d]--;  
    totIN--;  
    if (sospMoto[altradir(d)]>0 && autoIN[d]==0)  
        codaMoto[altradir(d)].signal();  
    else if (sospMoto[d]>0)  
        codaMoto[d].signal();  
    else if (sospAuto[altradir(d)]>0 && motoIN[d]==0  
            && autoIN[d]==0)  
        //possibile inversione di direzione:  
        codaAuto[altradir(d)].signalAll();  
    else if (sospAuto[d]>0 && totIN+4<=MAX)  
        codaAuto[d].signal();
```

Soluzione: Monitor

```
public void esceAuto(int d) {  
    lock.lock();  
    autoIN[d]--;  
    totIN-=4;  
    if (sospMoto[altradir(d)]>0 && autoIN[d]==0)  
        codaMoto[altradir(d)].signalAll();  
    else if (sospMoto[d]>0)  
        codaMoto[d].signalAll(); //1 auto vale 4 moto  
    if (sospAuto[altradir(d)]>0 && motoIN[d]==0  
        && autoIN[d]==0)  
        codaAuto[altradir(d)].signalAll();  
    else if (sospAuto[d]>0)  
        codaAuto[d].signal();  
    lock.unlock();  
}  
}
```

Esercizio 1 (1/3)

Si consideri una mostra di pittura che si svolge in una sala dedicata, all'interno di un grande centro espositivo. La mostra è frequentata da tre tipi di utenti:

- **Visitatore Singolo**: una singola persona che entra nella sala, visita la mostra e poi esce;
- **Scolaresca**: un gruppo di 30 persone che entrano nella sala, visitano la mostra e poi escono;
- **Addetto**: una singola persona dello staff con l'incarico di presidiare la sala che può ciclicamente entrare, presidiare la sala per un tempo arbitrario e poi uscire dalla sala;

L'**accesso** e l'**uscita** avvengono attraverso un **corridoio** che conduce all'unica porta di accesso della sala; il corridoio viene percorso dagli utenti sia in direzione IN (per entrare nella sala), sia in direzione OUT (per uscire dalla sala).

Esercizio 1 (2/3)

Accesso e Uscita dalla sala:

Per evitare situazioni di eccessivo assembramento **la sala ha una capacità limitata pari a MaxS** persone (visitatori, membri di scolaresche e addetti) oltre la quale non sarà consentito l'accesso a nessun utente.

Si assume, inoltre, che l'uscita di un addetto A dalla sala non possa avvenire se A è il solo addetto a presidiare la sala in quel momento.

Vincoli sul corridoio:

Per motivi di sicurezza il corridoio può essere percorso al più da NC persone.

Inoltre, poiché il corridoio è stretto, non è consentito il transito contemporaneo nel corridoio di scolaresche in direzioni opposte.

Esercizio 1 (3/3)

Realizzare un'applicazione concorrente in Java basata sul monitor nella quale ogni utente (visitatore singolo, scolaresca o addetto) sia rappresentato da un thread distinto.

La politica di sincronizzazione dovrà tenere in considerazione tutti i vincoli dati, ed inoltre:

- dovrà dare la **precedenza agli utenti in uscita** dalla sala;
- **in uscita:**
 - le scolaresche dovranno avere la precedenza sui visitatori singoli.
 - i visitatori singoli avranno la precedenza sugli addetti
- **in entrata:**
 - gli addetti avranno la precedenza sui visitatori singoli
 - i visitatori singoli dovranno avere la precedenza sulle scolaresche.

Impostazione

Quali thread?

- thread iniziale
- visitatori singoli
- scolaresche
- addetti

Quale risorsa comune?

- la **mostra**, cioè la sala e il corridoio

Associamo alla risorsa un "monitor", che controlla gli accessi e le uscite in base alla specifica politica di accesso. La sincronizzazione viene realizzata mediante **variabili condizione**.

Comportamento threads

Comportamento di ogni Visitatore (singolo o scolaresca):

Ogni visitatore/scolaresca si comporta come segue:

1. **Imbocca il corridoio in direzione IN;**
2. Percorre il corridoio (direzione IN)
3. **Esce dal corridoio in direzione IN** per entrare nella sala;
4. Visita la mostra
5. **Imbocca il corridoio in direzione OUT** uscendo dalla sala;
6. Percorre il corridoio (direzione OUT)
7. **Esce dal corridoio in direzione OUT.**

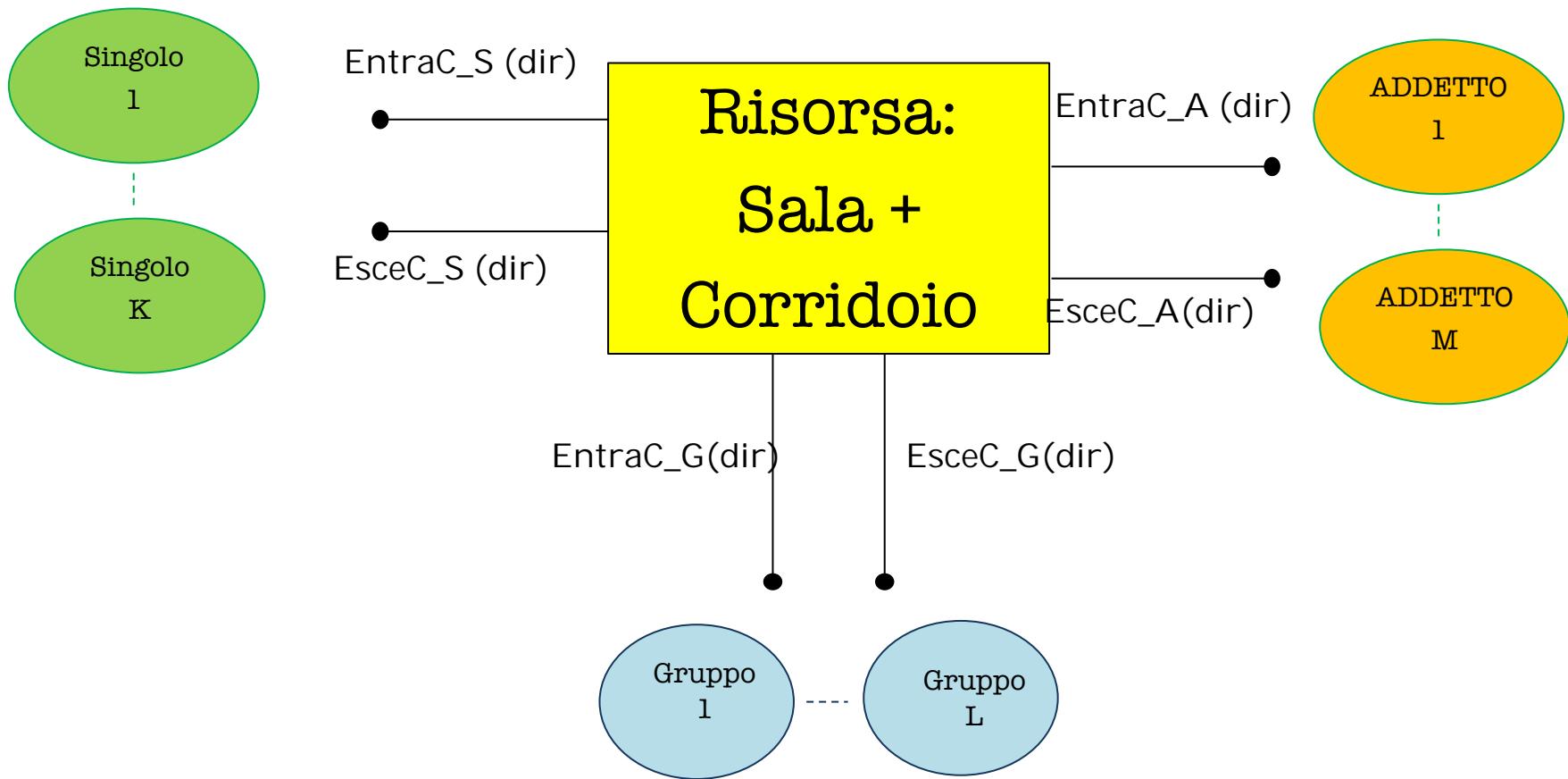
Comportamento di ogni Addetto:

ripete ciclicamente le seguenti fasi

1. **Imbocca il corridoio in direzione IN;**
2. Percorre il corridoio (direzione IN)
3. **Esce dal corridoio in direzione IN** per entrare nella sala;
4. Sta nella sala per presidiare la mostra
5. **Imbocca il corridoio in direzione OUT** uscendo dalla sala;
6. Percorre il corridoio (direzione OUT)
7. **Esce dal corridoio in direzione OUT.**

Impostazione

- Quali sono i thread? → **scolaresche, singoli e addetti**
- Qual è la risorsa condivisa? → **corridoio+sala**



Struttura dei thread

```
public class Singolo extends Thread{  
    private Monitor M;  
  
    public Singolo(...){ // costruttore..  
    }  
  
    public void run(){  
        ...  
        M.EntraC_S(IN);  
        <percorre corridoio>  
        M.EsceC_S(IN);  
        <visita Mostra>  
        M.EntraC_S(OUT);  
        <percorre corridoio in uscita>  
        M.EsceC_S(OUT);  
    }  
}
```

Struttura dei thread

```
public class Scolaresca extends Thread{  
    private Monitor M;  
  
    public Scolaresca(...){ // costruttore..  
    }  
  
    public void run(){  
        ...  
        M.EntraC_G(IN);  
        <percorre corridoio>  
        M.EsceC_G(IN);  
        <visita Mostra>  
        M.EntraC_G(OUT);  
        <percorre corridoio in uscita>  
        M.EsceC_G(OUT);  
    }  
}
```

Struttura dei thread

```
public class Addetto extends Thread{  
    private Monitor M;  
  
    public Scolaresca(...){ // costruttore..  
    }  
  
    public void run(){  
        while (...)  
        {            M.EntraC_A(IN);  
                <percorre corridoio>  
                M.EsceC_A(IN);  
                <presidia Mostra>  
                M.EntraC_A(OUT);  
                <percorre corridoio in uscita>  
                M.EsceC_A(OUT);  
        }  
    }  
}
```

Monitor: corridoio+sala

Variabili di stato:

Per il corridoio:

quanti addetti, singoli e gruppi in ogni direzione sul ponte

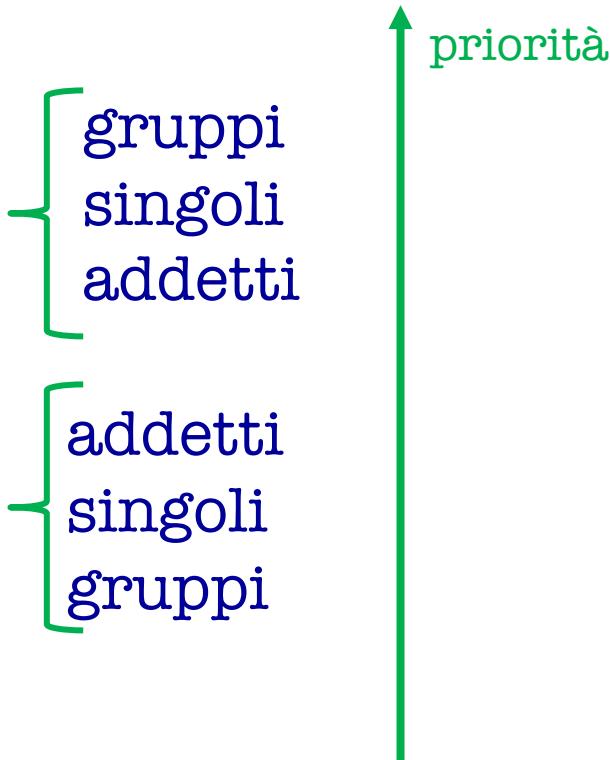
Per la sala:

quanti visitatori in sala, quanti addetti.

Politica di Sincronizzazione: scala delle priorità

Priorità - accesso al corridoio:

- in uscita dalla sala
- in entrata nell'isola



→ 6 livelli di priorità

Politica di Sincronizzazione

Un thread **Singolo** si sospende entrando nel corridoio:

Direzione IN (verso la sala):

- se il corridoio è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità)
- se la sala è piena (necessario, altrimenti il corridoio si potrebbe riempire di processi in attesa di entrare nella sala..)

Direzione OUT (uscita dalla sala):

- se il corridoio è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità..)

Politica di Sincronizzazione

Un thread **Scolaresca** si sospende entrando nel corridoio:

Direzione IN (verso la sala):

- se il corridoio è pieno
- se c'è almeno una scolaresca nel corridoio in dir OUT
- se c'è un processo più prioritario in attesa (v. scala priorità)
- se nella sala non c'è posto per una scolaresca

Direzione OUT (uscita dalla sala):

- se il corridoio è pieno
- se c'è almeno una scolaresca nel corridoio in dir IN
- se c'è un processo più prioritario in attesa (v. scala priorità..)

Politica di Sincronizzazione

Un thread **Addetto** si sospende **entrando nel Corridoio**:

Direzione IN (verso la sala):

- se il corridoio è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità)
- se la sala è piena

Direzione OUT (uscita dalla sala):

- se non ci sono altri addetti in sala oltre a lui
- se il corridoio è pieno
- se c'è un processo più prioritario in attesa (v. scala priorità...)

Esercizio 2

Estendere il problema dell'es.1 con scolaresche di consistenza numerica variabile:

ogni gruppo ha una numerosità arbitraria (al minimo 2, al massimo 30 persone).

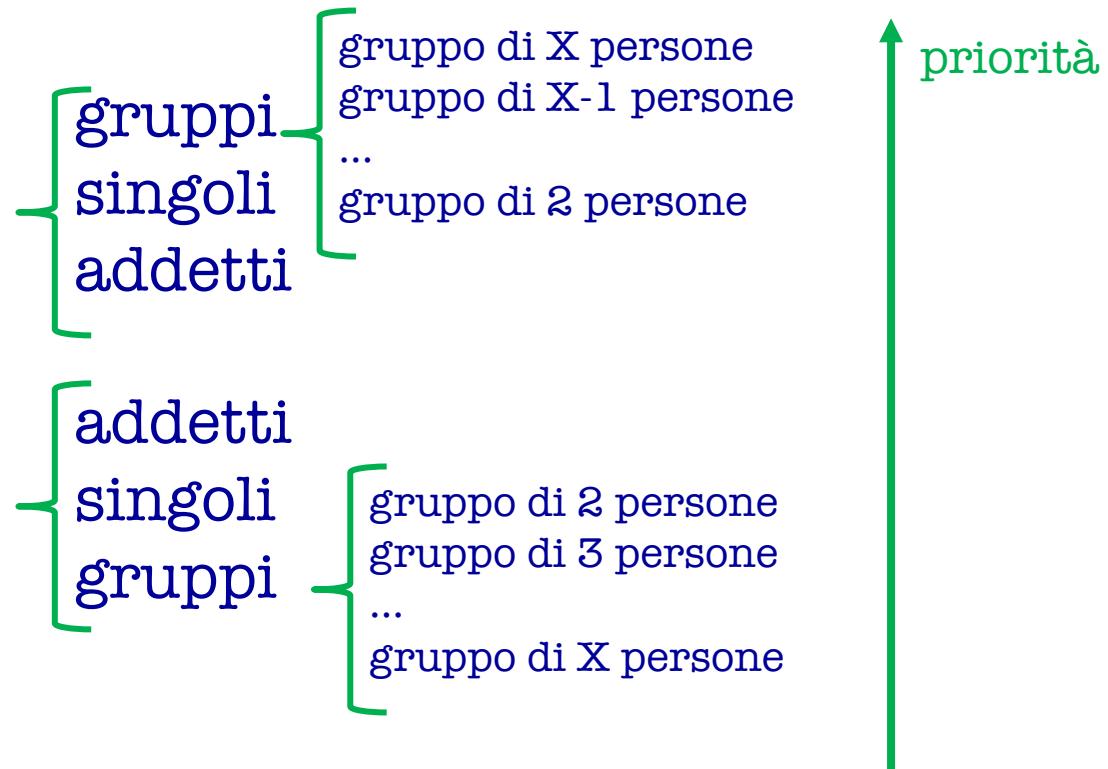
Si progetti una politica di gestione che tenga conto dei vincoli dati e che, inoltre:

- Nell'accesso al corridoio si dia sempre la precedenza agli utenti in uscita dalla sala.
- Nella direzione di entrata nell'isola: gli addetti abbiano la precedenza sui visitatori singoli; i singoli abbiano la precedenza sui gruppi; tra i gruppi **venga data la precedenza ai gruppi meno numerosi**.
- Nella direzione di uscita dall'isola: i gruppi abbiano la precedenza rispetto ai singoli; i singoli abbiano la precedenza sugli addetti. Tra i gruppi, **venga data la precedenza ai gruppi più numerosi**.

Politica di Sincronizzazione: scala delle priorità

Priorità - accesso al corridoio (X: massima numerosità gruppo):

- in uscita dalla sala
- in entrata nell'isola



→ $4 + 2 * (X-1)$ livelli di priorità

X=30 → $4 + 2 * 29 = 62$ livelli di priorità

Interazione tra Processi

Processi interagenti

Classificazione:

□ processi interagenti/indipendenti:

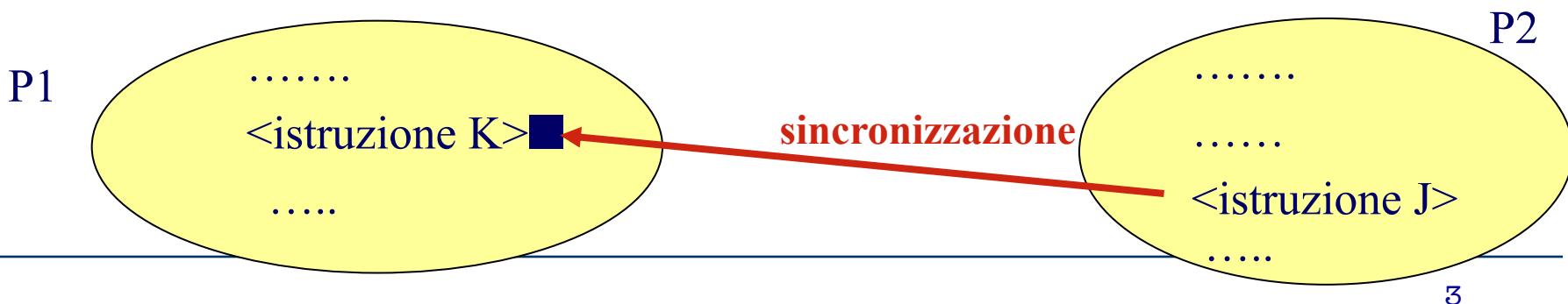
- due processi sono **interagenti** se l'esecuzione di un processo è in influenzata dall'esecuzione dell'altro processo o viceversa.
- due processi sono **indipendenti** se l'esecuzione di ognuno non è in alcun modo influenzata dall'altro.

□ processi interagenti:

- **cooperanti**: i processi interagiscono **volontariamente** per raggiungere obiettivi comuni (fanno parte della stessa applicazione)
- **in competizione**: i processi, in generale, non fanno della stessa applicazione, ma interagiscono **indirettamente**, per l'acquisizione di risorse comuni.

Processi Interagenti

- L'interazione puo` avvenire mediante due **meccanismi**:
 - ✓ **Comunicazione**: scambio di informazioni tra i processi interagenti.
 - ✓ **Sincronizzazione**: impostazione di vincoli temporali sull'esecuzione dei processi.
Ad esempio, l'istruzione K del processo P1 puo` essere eseguita soltanto dopo l'istruzione J del processo P2



Processi Interagenti

Realizzazione dell'interazione: dipende dal modello di processo:

- **modello ad ambiente locale (processi pesanti):** non c'è condivisione di variabili
 - la **Comunicazione** avviene attraverso **scambio di messaggi**
 - la **Sincronizzazione** avviene attraverso **scambio di eventi** (es: *segnali*)

I meccanismi di comunicazione/sincronizzazione vengono **realizzati dal sistema operativo**

- **modello ad ambiente globale (thread):** possibilità di condividere variabili
 - **Sincronizzazione:** strumenti di sincronizzazione (es. lock, semafori..)
 - **Comunicazione:** variabili condivise + strumenti di sincronizzazione (es.*semafori*)

Solo i meccanismi di sincronizzazione vengono **realizzati dal sistema operativo.** (la **comunicazione** viene realizzata dal **programmatore**)

Comunicazione

Scambio di messaggi

Facciamo riferimento al **modello ad ambiente locale**:

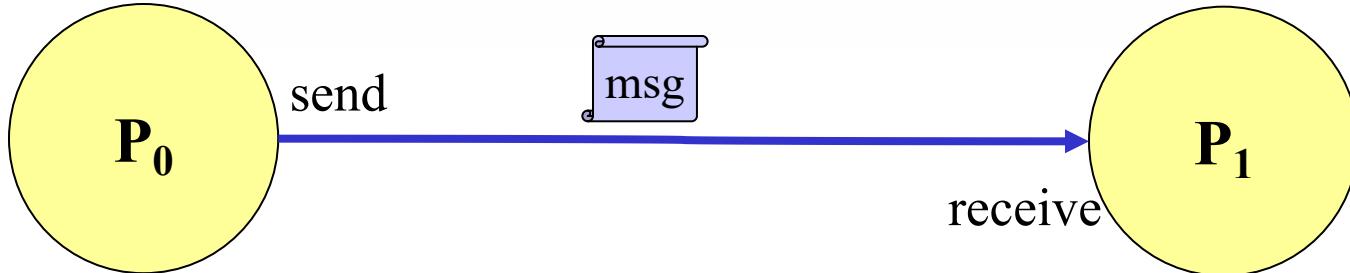
- non vi è memoria condivisa
- i processi possono interagire (*cooperano/ competono*)
 - mediante scambio di messaggi: *comunicazione*
 - mediante *scambio di eventi (o segnali)*: *sincronizzazione*

➤ Il Sistema Operativo offre meccanismi a supporto della comunicazione tra processi (*Inter Process Communication, o IPC*).

Operazioni Necessarie:

- ***send***: spedizione di messaggi da un processo ad altri
- ***receive***: ricezione di messaggi

Scambio di messaggi



Lo scambio di messaggi avviene mediante un **canale di comunicazione**.

Meccanismi di comunicazione tra processi

Aspetti caratterizzanti:

- tipo della comunicazione:
 - » diretta o indiretta
 - » simmetrica o asimmetrica
 - » bufferizzata o no
 - » ...
- caratteristiche del canale
 - » monodirezionale, bidirezionale
 - » uno-a-uno, uno-a-molti, molti-a-uno, molti-a-molti
 - » capacità
 - » modalità di creazione: automatica, non automatica
- caratteristiche del messaggio:
 - » dimensione
 - » tipo

Naming

In che modo viene specificata la destinazione di un messaggio?

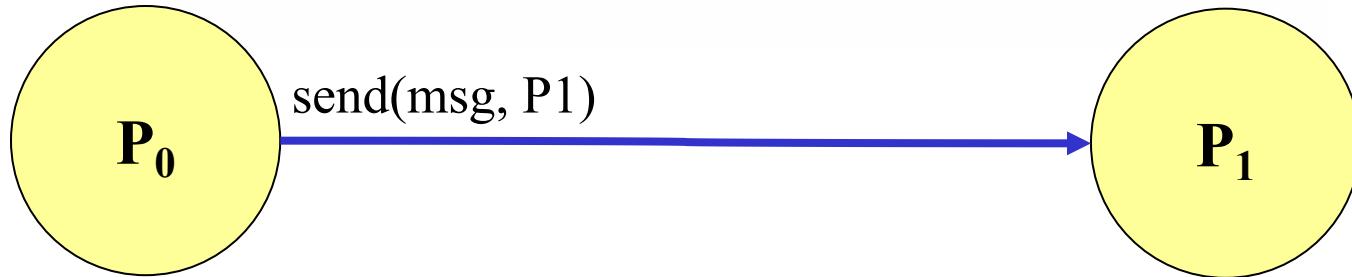
- **Comunicazione diretta:** al messaggio viene associato l'identificatore del processo destinatario (naming esplicito)

send(Proc, msg)

- **Comunicazione indiretta:** il messaggio viene indirizzato a una mailbox (contenitore di messaggi) dalla quale il destinatario preleverà il messaggio:

send(Mailbox, msg)

Comunicazione diretta



- Il canale è creato automaticamente tra i due processi che devono *conoscersi* reciprocamente:
 - canale punto-a-punto
 - canale bidirezionale:
 - p0: send(query, P1);
 - p1: send(answ, P0)
 - per ogni coppia di processi esiste un solo canale(<P0, P1>)

Esempio: Produttore & Consumatore

Processo produttore P:

```
pid C =....;  
main()  
{ msg M;  
  do  
  { produco (&M) ;  
    ...  
    send (C, M) ;  
  }while (!fine) ;  
}
```

Processo consumatore C:

```
pid P=....;  
main()  
{ msg M;  
  do  
  { receive (P, &M) ;  
    ...  
    consumo (M) ;  
  }while (!fine) ;  
}
```

Comunicazione simmetrica:

- il destinatario fa il ***naming*** esplicito del mittente

Comunicazione asimmetrica

Processo produttore P:

```
....  
main()  
{ msg M;  
  do  
  { produco (&M) ;  
    ...  
    send (C, M) ;  
  }while (!fine) ;  
}
```

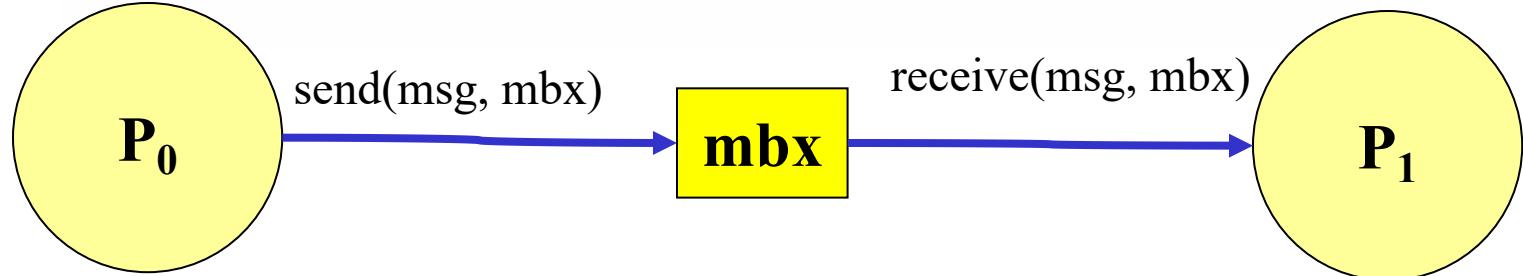
Processo consumatore C:

```
....  
main()  
{ msg M; pid id;  
  do  
  { receive (&id, &M) ;  
    ...  
    consumo (M) ;  
  }while (!fine) ;  
}
```

Comunicazione asimmetrica:

- il destinatario non è obbligato a conoscere l'identificatore del mittente: la variabile **id** raccoglie l'identificatore del mittente. (es. Modello client-server)

Comunicazione indiretta



- I processi cooperanti non sono tenuti a conoscersi reciprocamente e si scambiano messaggi depositandoli/prelevandoli da una mailbox *condivisa*.
- La **mailbox** è un canale utilizzabile da più processi che funge da contenitore dei messaggi.

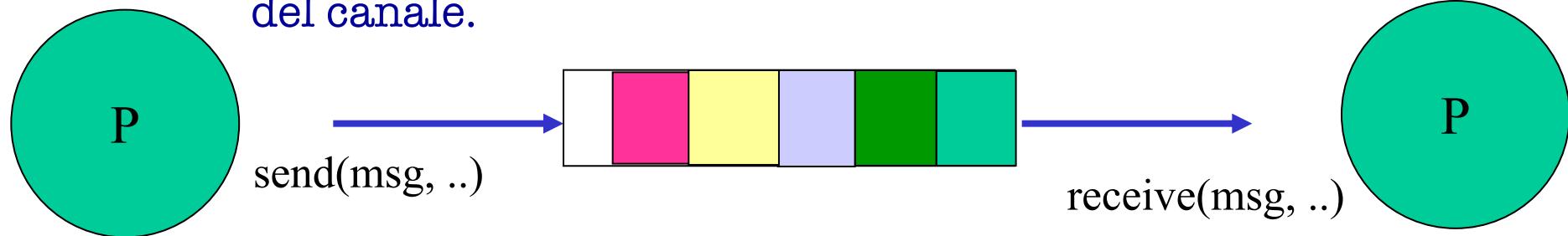
Comunicazione indiretta

Proprietà:

- ❑ il canale di comunicazione è rappresentato dalla **mailbox** (non viene creato automaticamente)
- ❑ il canale può essere associato a più di 2 processi:
 - ✓ **mailbox di sistema**: multi-a-multi (come individuare il processo destinatario di un messaggio?)
 - ✓ **mailbox del processo destinatario (porta)**: multi-a-uno
- ❑ per ogni coppia di processi possono esistere più canali (uno per ogni mailbox condivisa)

Buffering del canale

- Ogni canale di comunicazione è caratterizzato da una **capacità ($>= 0$)**: numero dei messaggi che è in grado di contenere contemporaneamente.
- Gestione secondo politica **FIFO**:
 - i messaggi vengono posti in una coda in attesa di essere ricevuti
 - la lunghezza massima della coda rappresenta la capacità del canale.



Buffering del canale

- **Capacità nulla:** non vi è accodamento perchè il canale non è in grado di gestire messaggi in attesa
 - processo mittente e destinatario devono **sincronizzarsi** all'atto di spedire (send) / ricevere (receive) il messaggio: comunicazione **sincrona** o **rendez vous**
 - **send e receive** possono essere **sospensive**:



Buffering del canale

- **Capacità non nulla (limitata)**: esiste un limite N alla dimensione della coda:
 - se la coda **non è piena**, un nuovo messaggio viene posto in fondo
 - se la coda **è piena**: la send è sospensiva
 - se la coda **è vuota**: la receive può essere sospensiva
- [**Capacità illimitata**: lunghezza della coda teoricamente infinita: non c'è possibilità di sospensione.]

Send sincrona/asincrona

Quindi, a seconda della capacità del canale, la send può essere sospensiva o no:

- **Canale a Capacità nulla: Send sincrona**

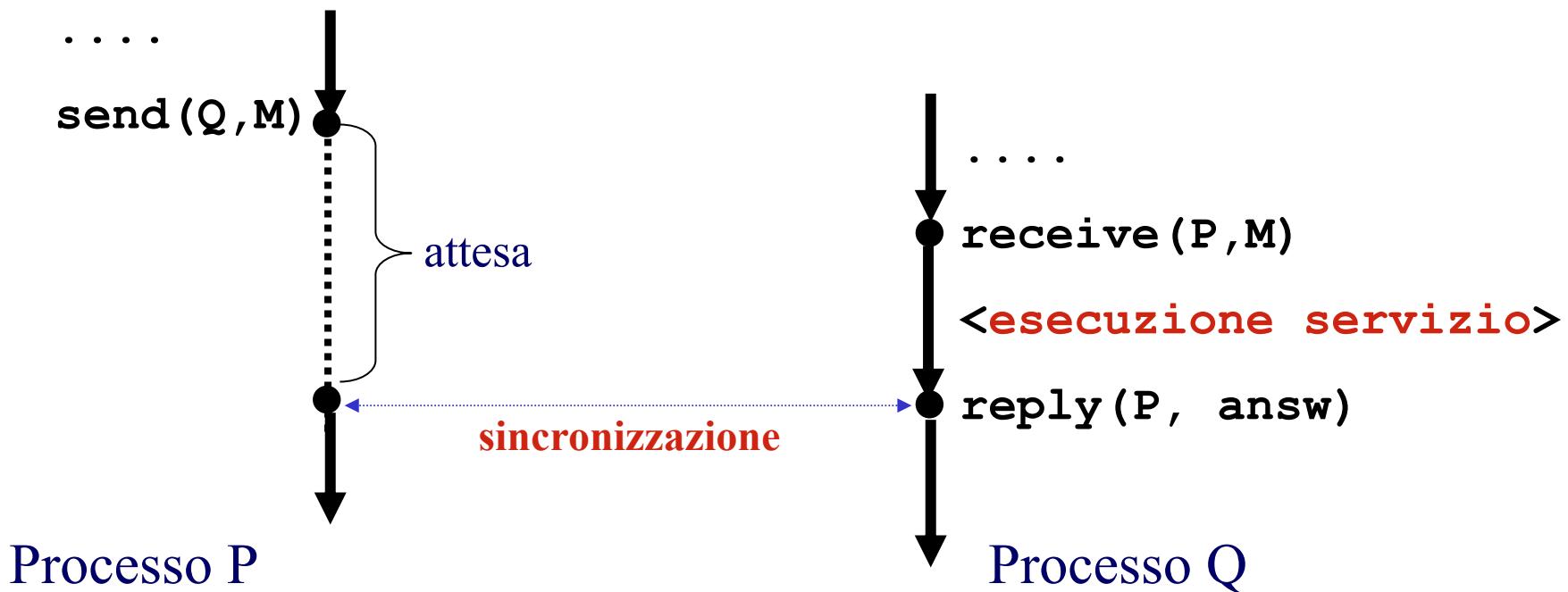
- Se il destinatario non è pronto per ricevere il messaggio, il mittente attende.

- **Canale a Capacità non nulla: Send asincrona**

- La send non è sospensiva: il mittente deposita il messaggio nel canale e continua la sua esecuzione.

Send con sincronizzazione estesa

- È un tipo di comunicazione sincrona, in cui il mittente si sospende fino a che il destinatario non restituisce una risposta (**reply**) al messaggio inviato:
 - il messaggio potrebbe richiedere l'esecuzione di un **servizio** (es. **Remote Procedure Call, RPC**)



Caratteristiche della comunicazione tra processi Unix

- Tra meccanismi di comunicazione:
 - **pipe**: comunicazione locale (nell'ambito della stessa gerarchia di processi)
 - **fifo**: comunicazione locale (anche tra processi di gerarchie diverse)
 - **socket**: comunicazione in ambiente distribuito (tra processi in esecuzione su nodi diversi di una rete)
- **Pipe**: comunicazione
 - » **indiretta** (senza naming esplicito)
 - » canale **unidirezionale multi-a-multi**
 - » **bufferizzata** (capacità limitata): possibilità di sospensione sia per mittenti che per destinatari.

Sincronizzazione

Sincronizzazione tra processi

La sincronizzazione permette di imporre vincoli sulle operazioni dei processi interagenti.

Ad Esempio:

Nella cooperazione:

- Per imporre un particolare ordine cronologico alle azioni eseguite dai processi interagenti.
- Per garantire che le operazioni di comunicazione avvengano secondo un ordine prefissato.
- Per notificare l'avvenimento di un evento

Nella competizione:

- Per garantire la mutua esclusione dei processi nell'accesso alla risorsa condivisa.
- Per realizzare politiche di accesso alle risorse condivise

Sincronizzazione tra processi nel modello ad ambiente locale

In questo modello non c'e` la possibilita` di condividere memoria:

- ❑ Gli accessi alle risorse "condivise" vengono controllati e coordinati dal sistema operativo.
- ❑ La sincronizzazione avviene mediante **meccanismi offerti dal sistema operativo** che consentono la notifica di “eventi” asincroni tra processi:

- Es. segnali unix



Sincronizzazione tra processi nel modello ad ambiente globale

Facciamo riferimento a processi che possono condividere variabili (**modello ad ambiente globale**, o a **memoria condivisa**):

In questo ambiente:

- **cooperazione**: lo scambio di messaggi avviene attraverso strutture dati condivise (ad es., mailbox)
- **competizione**: le risorse sono rappresentate da variabili condivise (ad esempio, puntatori a file). Ad es: Un oggetto condiviso non può essere acceduto da più di un processo alla volta (**problema della mutua esclusione**).

In entrambi i casi è necessario sincronizzare i processi per coordinarli nell'accesso alla memoria condivisa.

- > Il SO definisce alcuni **strumenti nella memoria comune** che consentono a ogni processo di aspettare il verificarsi di condizioni e di riattivare processi in attesa (ad esempio: **semafori, lock**).

Sincronizzazione tra processi in Unix: i segnali

Sincronizzazione tra processi

La sincronizzazione permette di imporre vincoli sull'ordine di esecuzione delle operazioni dei processi interagenti.

Unix adotta il **modello ad ambiente locale**: la sincronizzazione può realizzarsi mediante **segnali**

Segnale:

è un'**interruzione software**, che **notifica un evento in modo asincrono** al processo che la riceve.

Il processo destinatario rileva l'evento istantaneamente, qualunque sia il suo stato.

Segnali

Un segnale **notifica un evento**.

Eventi:

- generati da terminale (es. CTRL+C)
- generati dal kernel in seguito ad eccezioni HW
(violazione dei limiti di memoria, divisioni per 0, etc.)
- generati dal kernel in seguito a condizioni SW (time-out, scrittura su pipe chiusa, etc.)
- **generati da altri processi**

Segnali Unix

Un segnale può essere inviato:

- dal kernel a processi utente
- da un processo utente ad altri processi utente

(Es: comando **kill**)

Quando un processo riceve un segnale, può comportarsi in **tre modi diversi**:

1. **gestire** il segnale con una funzione **handler** definita dal programmatore
2. **eseguire** un'azione predefinita dal S.O. (azione di **default**)
3. **ignorare** il segnale (**nessuna reazione**)

Nei primi due casi, il processo **reagisce in modo asincrono** al segnale:

1. **interruzione** dell'esecuzione
2. **esecuzione** dell'azione associata (**handler** o **default**)
3. **ritorno** alla prossima istruzione del codice del processo interrotto

Segnali Unix

Per ogni versione di Unix esistono vari tipi di segnale (in Linux, 32 segnali), ognuno identificato da un intero.

Ogni segnale, è associato a un particolare evento e prevede una specifica **azione di default**.

È possibile riferire i segnali con identificatori simbolici (**SIGxxx**):

SIGKILL, SIGSTOP, SIGUSR1, etc.

L'associazione tra nome simbolico e intero corrispondente (che dipende dalla versione di Unix) è specificata nell'header file **<signal.h>**.

Segnali Unix (linux): signal.h

```
#define SIGHUP 1          /* Hangup (POSIX). Action: exit */
#define SIGINT 2           /* Interrupt (ANSI). Action: exit (^C) */
#define SIGQUIT 3          /* Quit (POSIX). Action: exit, core dump */
#define SIGILL  4           /* Illegal instr. (ANSI). Action: exit,core dump */

...
#define SIGKILL 9           /* Kill, unblockable (POSIX). Action: exit*/
#define SIGUSR1 10          /* User-defined signal 1 (POSIX). Action: exit*/
#define SIGSEGV 11          /* Segm. violation (ANSI). Act: exit,core dump */
#define SIGUSR2 12          /* User-defined signal 2 (POSIX).Act: exit */
#define SIGPIPE 13          /* Broken pipe (POSIX).Act: exit */
#define SIGALRM 14          /* Alarm clock (POSIX). Act: exit */
#define SIGTERM 15          /* Termination (ANSI). Act:exit*/

...
#define SIGCHLD 17          /* Child status changed (POSIX).Act: ignore */
#define SIGCONT 18           /* Continue (POSIX).Act. ignore */
#define SIGSTOP 19           /* Stop, unblockable (POSIX). Act: stop */
...
```

Quali segnali nel mio sistema?

Comando di shell kill con opzione -l:

```
anna@www-lia:~$ kill -l
```

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	..		

Per maggiori dettagli:

```
$ man signal
```

Gestione dei segnali

Si è visto che, **in generale**, quando un processo riceve un segnale, può comportarsi in **3 modi diversi**:

1. **gestire** il segnale con una funzione **handler** definita dal programmatore
2. **eseguire** un'azione predefinita dal S.O. (azione di **default**)
3. **ignorare** il segnale

Casi particolari: esistono 2 segnali che non possono essere gestiti esplicitamente dai processi:

SIGKILL e SIGSTOP non sono **nè intercettabili, nè ignorabili.**

- qualunque processo, alla ricezione di SIGKILL o SIGSTOP esegue sempre l'**azione di default**.

System call signal

Ogni processo può gestire esplicitamente un segnale utilizzando la system call **signal**:

```
void (* signal(int sig, void (*func)()))(int);
```

- **sig** è l'**intero** (o il nome simbolico) che individua il segnale da gestire
- il parametro **func** è un **puntatore a una funzione** che indica l'azione da associare al segnale; in particolare **func** può:
 - » puntare alla routine di gestione dell'interruzione (**handler**)
 - » valere **SIG_IGN** (nel caso di segnale ignorato)
 - » valere **SIG_DFL** (nel caso di azione di default)
- **ritorna un puntatore a funzione:**
 - » al precedente gestore del segnale
 - » la costante **SIG_ERR**, nel caso di errore

signal

Ad esempio:

```
#include <signal.h>
void gestore(int);
...
main()
{...
signal(SIGUSR1, gestore); /*SIGUSR1 gestito */
...
signal(SIGUSR1, SIG_DFL); /*USR1 torna a default */
signal(SIGKILL, SIG_IGN); /*errore! SIGKILL non è
                           ignorabile */
...
}
```

Routine di gestione del segnale *(handler)*:

Caratteristiche:

- ❑ l'*handler* prevede sempre **un parametro formale** di tipo **int** che rappresenta **il numero del segnale effettivamente ricevuto**.
- ❑ l'*handler* non restituisce alcun risultato

```
void handler(int signum)  
{ . . .  
    . . .  
    return;  
}
```

Routine di gestione del segnale *(handler)*:

Struttura del programma:

```
#include <signal.h>
void handler(int signum)
{<istruzioni per la gestione del
segnale>
return;
}

main()
{
    ...
    signal(SIGxxx, handler);
    ...
}
```

Esempio: parametro del gestore

```
/* file segnali1.c */
#include <signal.h>
void handler(int);

main()
{ if (signal(SIGUSR1, handler)==SIG_ERR)
    perror("prima signal non riuscita\n");
  if (signal(SIGUSR2, handler)==SIG_ERR)
    perror("seconda signal non riuscita\n");
  for (;;) ;
}

void handler (int signum)
{ if (signum==SIGUSR1) printf("ricevuto sigusr1\n");
  else if (signum==SIGUSR2) printf("ricevuto
  sigusr2\n");
}
```

Prerequisito: comando kill

Il comando di shell **kill** serve ad inviare segnali ai processi.

Sintassi:

```
$ kill -signal_number pid ...
```

Esempio:

```
bash-3.2$ ./prova&
```

```
[1] 40151
```

```
bash-3.2$ kill -9 40151
```

```
bash-3.2$
```

```
[1]+  Killed: 9    ./prova
```

```
bash-3.2$
```

creazione in **background** del processo che esegue prova (pid 40151)

invio **segnalet n.9** (SIGKILL) al processo con pid 40151

il processo con pid 40151 è terminato

Esempio: esecuzione & comando kill

```
anna@lab3-linux:~/esercizi$ vi segnalil.c
anna@lab3-linux:~/esercizi$ gcc segnalil.c
anna@lab3-linux:~/esercizi$ a.out&
[1] 313
anna@lab3-linux:~/esercizi$ kill -SIGUSR1 313
anna@lab3-linux:~/esercizi$ ricevuto sigusr1

anna@lab3-linux:~/esercizi$ kill -SIGUSR2 313
anna@lab3-linux:~/esercizi$ ricevuto sigusr2

anna@lab3-linux:~/esercizi$ kill -9 313
anna@lab3-linux:~/esercizi$
[1]+  Killed                  a.out
anna@lab3-linux:~/esercizi$
```

Esempio: gestore del SIGCHLD

- **SIGCHLD** è il segnale che il kernel invia a un processo padre quando un figlio termina.
[**NB**: L'azione di **default** per **SIGCHLD** è **SIG_IGN**]
- È possibile svincolare il padre da un'attesa esplicita della terminazione del figlio, mediante un'apposita funzione **handler** per la gestione di **SIGCHLD**:
 - la funzione **handler** verrà attivata in modo **asincrono** alla ricezione del segnale
 - handler chiamerà la **wait** con cui il padre potrà raccogliere ed eventualmente gestire lo stato di terminazione del figlio

Esempio: gestore del SIGCHLD

```
#include <signal.h>
#include <stdio.h>
void handler(int);

main()
{ int PID, i;
  signal(SIGCHLD,handler);
  PID=fork();
  if (PID>0) /* padre */
  {
    for (i=0; i<10000000; i++) /* attività del padre...*/
      printf("il padre sta lavorando...\n");
    exit(0); }
  else /* figlio */
  { signal(SIGCHLD,SIG_DFL);
    for (i=0; i<1000; i++); /* attività del figlio...*/
    exit(1); }
}
```

Esempio: gestore del SIGCHLD

```
void handler (int signum)
{ int status;
  wait(&status);
  if ((char)status==0)
    printf("term. volontaria con stato %d", status>>8);
  else printf("term. involontaria per segnale
%d\n", (char)status);
}
```

Segnali & fork

Le associazioni segnali-azioni vengono registrate nella **User Structure** del processo.

Sappiamo che:

- una **fork** copia la **User Structure** del padre nella **User Structure** del figlio
- padre e figlio condividono lo stesso codice
quindi
- il figlio eredita dal padre le informazioni relative alla gestione dei segnali:
 - **ignora** gli stessi **segnali ignorati dal padre**
 - **gestisce** con le stesse funzioni gli stessi **segnali gestiti dal padre**
 - i **segnali a default** del figlio sono gli **stessi del padre**

👉 successive signal del figlio modificano solo la user structure del figlio e pertanto non hanno effetto sulla gestione dei segnali del padre.

Segnali & exec

Sappiamo che:

- una **exec** sostituisce codice e dati del processo che la chiama
- in seguito a una chiamata a exec il contenuto della **User Structure** viene **mantenuto**, tranne le **informazioni legate al codice** del processo (ad esempio, le funzioni di gestione dei segnali, che dopo l'**exec** non sono più visibili!)

quindi

- dopo un'**exec**, un processo:
 - **ignora** gli stessi **segnali ignorati prima** di exec
 - i **segnali a default** rimangono a **default**

ma
 - i **segnali che prima erano gestiti**, vengono riportati a **default**

Esempio

```
/* file segnali2.c */
#include <signal.h>

main()
{
    signal(SIGINT, SIG_IGN);
    execl("/bin/sleep", "sleep", "30", (char *)0);

}
```

N.B. Il comando: sleep N
mette nello stato *sleeping* il processo per N secondi

Esempio: esecuzione

```
anna@lab3-linux:~/esercizi$ gcc segnali2.c
anna@lab3-linux:~/esercizi$ a.out&
[1] 500
anna@lab3-linux:~/esercizi$ kill -SIGINT 500
anna@lab3-linux:~/esercizi$ kill -9 500
anna@lab3-linux:~/esercizi$
[1]+  Killed                  a.out
anna@lab3-linux:~/esercizi$
```

System call kill

I processi possono inviare segnali ad altri processi con la system call **kill**:

```
int kill(int pid, int sig);
```

- **sig** è l'intero (o il nome simbolico) che individua il segnale da gestire
- il parametro **pid** specifica il destinatario del segnale:
 - » **pid > 0**: l'intero è il pid dell'unico processo destinatario
 - » **pid=0**: il segnale è spedito a tutti i processi appartenenti al **gruppo** del mittente
 - » **pid < -1**: il segnale è spedito a tutti i processi con groupId uguale al valore assoluto di pid
 - » **pid== -1**: vari comportamenti possibili (Posix non specifica)

kill: esempio:

```
#include <stdio.h>
#include <signal.h>
int cont=0;

void handler(int signo)
{ printf ("Proc. %d: ricevuti n. %d segnali %d\n",
    getpid(),cont++, signo);
}

main ()
{int pid;
 signal(SIGUSR1, handler);
 pid = fork();
 if (pid == 0) /* figlio */

    for (;;);

else /* padre */
    for(;;) kill(pid, SIGUSR1);
}
```

Segnali: altre system call

sleep:

`unsigned int sleep(unsigned int N)`

- provoca la sospensione del processo per N secondi (al massimo)
- se il processo riceve un segnale durante il periodo di sospensione, viene risvegliato *prematuramente*
- ritorna:
 - » 0, se la sospensione non è stata interrotta da segnali
 - » se il risveglio è stato causato da un segnale al tempo N_s , `sleep` restituisce il numero di secondi non utilizzati dell'intervallo di sospensione ($N - N_s$).

Esempio

```
/* provasleep.c*/
#include <signal.h>
void stampa(int signo)
{ printf("risvegliato dal segnale %d !!\n",
signo);
}
main()
{
    int k;
    signal(SIGUSR1, stampa);
    k=sleep(1000);
    printf("Mancavano ancora %d sec.. \n", k);
    exit(0);
}
```

Esecuzione

```
bash-2.05$ gcc -o pr provasleep.c
bash-2.05$ pr&
[1] 9950
bash-2.05$ kill -SIGUSR1 9950
bash-2.05$ risvegliato dal segnale 10!!
Mancavano ancora 992 sec..
```

Segnali: altre system call

alarm:

`unsigned int alarm(unsigned int N)`

- Imposta un timer che dopo N secondi invierà al processo il segnale **SIGALRM**;
- ritorna:
 - » 0, se non vi erano time-out impostati in precedenza
 - » il numero di secondi mancante allo scadere del time-out precedente

NB: la alarm **non è sospensiva**;
l'azione di **default** associata a SIGALRM è la terminazione.

Segnali: altre system call

pause:

```
int pause(void)
```

- **sospende** il processo fino alla ricezione di un qualunque segnale
- ritorna -1 (`errno = EINTR`)

Esempio

Due processi (padre e figlio) si sincronizzano alternativamente mediante il segnale SIGUSR1 (gestito da entrambi con la funzione *handler*):



```
int ntimes = 0;  
void handler(int signo)  
{printf ("Processo %d ricevuto #%-d volte il segnale %d\n",  
       getpid(), ++ntimes, signo);  
}
```

```
main ()
{
    int pid, ppid;
    signal(SIGUSR1, handler);
    if ((pid = fork()) < 0) /* fork fallita */
        exit(1);
    else if (pid == 0) /* figlio*/
    {
        ppid = getppid(); /* PID del padre */
        for (;;)
        {
            printf("FIGLIO %d\n", getpid());
            sleep(1);
            kill(ppid, SIGUSR1);
            pause();
        }
    }
    else /* padre */
        for(;;) /* ciclo infinito */
    {
        printf("PADRE %d\n", getpid());
        pause();
        sleep(1);
        kill(pid, SIGUSR1);
    }
}
```

Test:

```
anna$ ./provasegnali  
PADRE 42300  
FIGLIO 42301  
Processo 42300 ricevuto #1 volte il segnale 30  
PADRE 42300  
Processo 42301 ricevuto #1 volte il segnale 30  
FIGLIO 42301  
Processo 42300 ricevuto #2 volte il segnale 30  
PADRE 42300  
Processo 42301 ricevuto #2 volte il segnale 30  
FIGLIO 42301  
Processo 42300 ricevuto #3 volte il segnale 30  
PADRE 42300  
Processo 42301 ricevuto #3 volte il segnale 30  
FIGLIO 42301  
Processo 42300 ricevuto #4 volte il segnale 30  
PADRE 42300  
Processo 42301 ricevuto #4 volte il segnale 30  
...
```

Gestione di segnali con handler

- Non sempre l'associazione ***segnale/handler*** è durevole:
 - alcune implementazioni di Unix (BSD, SystemV r.3 e seg.), prevedono che l'azione rimanga installata anche dopo la ricezione del segnale.
 - in alcune realizzazioni (SystemV, prime versioni), invece, dopo l'attivazione dell'handler ripristina automaticamente l'azione di default. In questi casi, per riagganciare il segnale all'handler:

```
main()
{
    ...
    signal(SIGUSR1, f);

    ...
}
```

```
void f(int s)
{
    signal(SIGUSR1, f);

    ...
}
```

Affidabilità dei segnali

Cosa succede se
qui arriva un nuovo
segnale?

```
void handler(int s)
{ signal(SIGUSR1, handler);
  printf("Processo %d: segnale %d\n", getpid(), s);
  ... }
```

cosa succede se arriva il segnale durante
l'esecuzione dell'handler? Tre alternative:

- **innestamento** delle routine di gestione
- **perdita** del segnale
- **accodamento** dei segnali (segnali ***reliable, BSD 4.2***)

Interrompibilità di System Calls

- System Call: possono essere classificate in
 - **slow** system call: possono richiedere tempi di esecuzione non trascurabili perchè possono causare periodi di attesa (es: lettura da un dispositivo di I/O lento, wait, sleep ecc.)
 - system call **non slow** (es. getpid, signal, ecc.)
- una **slow** system call è interrompibile da un segnale; in caso di interruzione:
 - » ritorna -1
 - » **errno** vale EINTR
- possibilità di ri-esecuzione della system call:
 - » automatica (BSD 4.3)
 - » non automatica, ma comandata dal processo (in base al valore di **errno** e al valore restituito)

Segnali in Linux

- **Persistenza dell'handler:** alla fine dell'esecuzione di un handler definito dall'utente, il sistema si occupa di **reinstallarlo automaticamente**
- Se, durante l'esecuzione di un handler, **arriva un secondo segnale uguale** a quello che ha causato la sua esecuzione, il segnale viene **accodato** e gestito una volta terminato il primo handler.
- Segnali che arrivano mentre c'è un segnale **uguale** accodato vengono "**accorpati**" in uno: **i.e.** solo un segnale verrà consegnato al processo (e.g. molte chiamate a `kill()` eseguite in tempi ravvicinati).

Il File System

Il file system

E' quella componente del SO che fornisce i **meccanismi di accesso e memorizzazione** delle informazioni (programmi e dati) **allocate in memoria di massa**

Realizza i concetti astratti

- di **file**: **unità logica** di memorizzazione
 - di **direttorio**: **insieme di file** (e direttori)
 - di **partizione**: insieme di file associato ad un **particolare dispositivo fisico** (o porzione di esso)
- Le **caratteristiche** di file, directory e partizione sono **del tutto indipendenti** da natura e tipo di dispositivo utilizzato.

Organizzazione del File System

La struttura di un file system essere rappresentata da un insieme di **componenti** organizzate in vari livelli:



Hardware: memoria secondaria

Organizzazione del file system

- **Struttura logica:** presenta alle applicazioni una **visione astratta** delle informazioni memorizzate, basata su **file, directory, partizioni**, ecc.. Realizza le operazioni di gestione di file e directory: copia, cancellazione, spostamento, ecc.
- **Accesso:** definisce e realizza i meccanismi per accedere al contenuto dei file; in particolare:
 - Definisce l'unità di trasferimento da/verso file: **record logico**
 - Realizza i **metodi di accesso** (sequenziale, casuale, ad indice)
 - Realizza i **meccanismi di protezione**
- **Organizzazione fisica:** rappresentazione di file e directory sul dispositivo:
 - **Allocazione** dei file sul dispositivo (unità di memorizzazione = **blocco**): mapping di record logici su blocchi. Vari metodi di allocazione.
 - **Rappresentazione** della struttura logica sul dispositivo.
- **Dispositivo Virtuale:** presenta una vista astratta del dispositivo, che appare come una sequenza di **blocchi**, ognuno di dimensione data costante.

Struttura logica

File

È un insieme di informazioni; ad es.:

- programmi
- dati (in rappresentazione binaria)
- dati (in rappresentazione testuale)
- ...
- Ogni file è ***individuato da (almeno) un nome simbolico*** mediante il quale può essere riferito (ad esempio, nell'invocazione di comandi o di system call)
- Ogni file è ***caratterizzato da un insieme di attributi***

Attributi del file

A seconda del SO, i file possono avere attributi diversi.

Soltamente:

- **tipo**: stabilisce l'appartenenza a una classe (eseguibili, testo, musica, non modificabili, ...)
- **indirizzo**: puntatore/i a memoria secondaria
- **dimensione**: numero di byte contenuti nel file
- **data e ora** (di creazione e/o di modifica)

In SO multiutente anche:

- **utente proprietario**
- **protezione**: **diritti di accesso** al file per gli utenti del sistema

Attributi del file

Descrittore del file:

è la struttura dati che contiene gli attributi del file

Ogni **descrittore** di file deve essere **memorizzato in modo persistente**:

- 👉 il SO mantiene **l'insieme dei descrittori di tutti i file** presenti nel file system in apposite strutture **in memoria secondaria** (ad es. UNIX: **i-list**)

Tipi di file: nomi ed estensioni

In alcuni SO,
l'estensione inclusa nel
nome di un file
rappresenta il suo tipo

NON è il caso di UNIX

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Directory (o directory)

Strumento per **organizzare i file all'interno del file system**:

- una directory può **contenere** più file
- è realizzata mediante una **struttura dati** che prevede un elemento per ogni file (o directory) in essa contenuto; essa associa al nome di ogni file le informazioni che consentono di localizzarlo in memoria di massa.

Operazioni sui direttori:

- **Creazione/cancellazione** di directory
- **Aggiunta/cancellazione** di file
- **Listing**: elenco di tutti i file contenuti nella directory
- **Attraversamento** della directory
- **Ricerca** di file in directory

Tipi di directory

La **struttura logica delle directory** può variare a seconda del SO

Schemi più comuni:

- a un livello**
- a due livelli**
- ad albero**
- a grafo aciclico**

Tipi di directory

Struttura a un livello: una sola directory per ogni file system



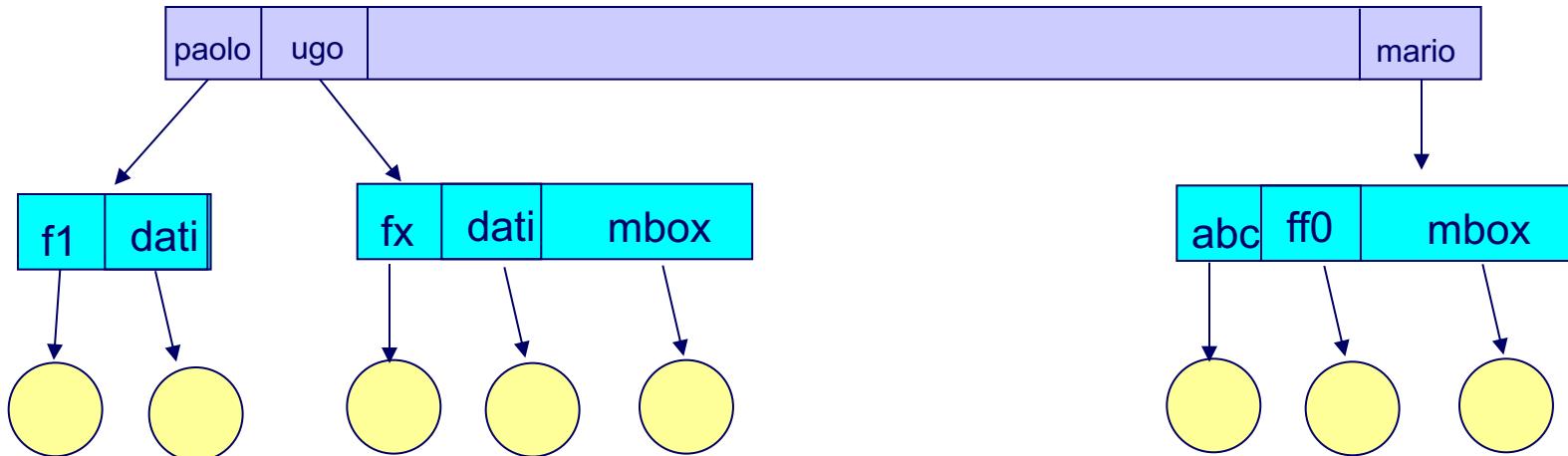
Problemi

- **unicità dei nomi**
- **multiutenza:** come separare i file dei diversi utenti?

Tipi di directory

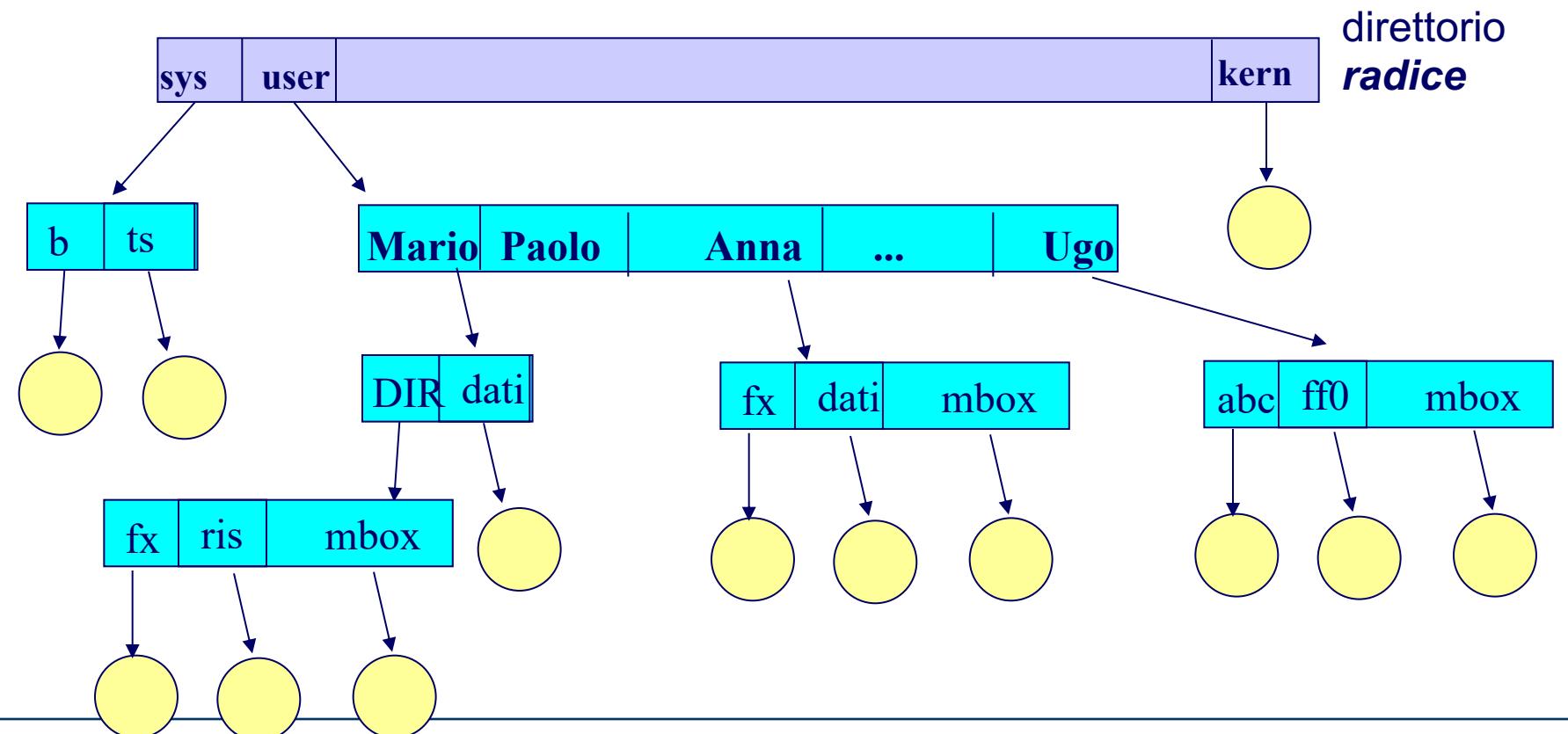
Struttura a due livelli

- primo livello (**directory principale**): contiene una directory per ogni utente del sistema
- secondo livello: **directory utenti** (a un livello)



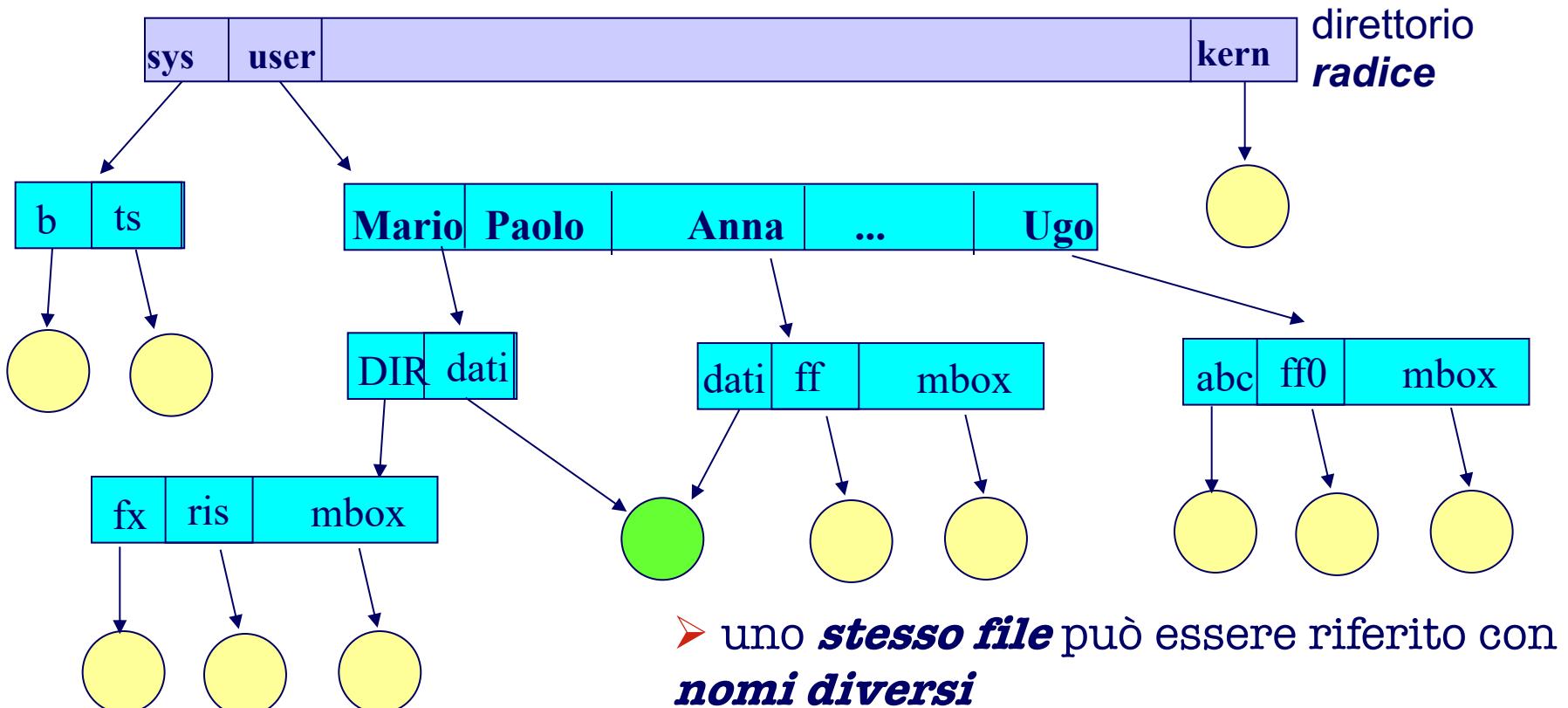
Tipi di directory

Struttura ad albero: organizzazione gerarchica a N livelli. Ogni direttorio può contenere file e altri direttori



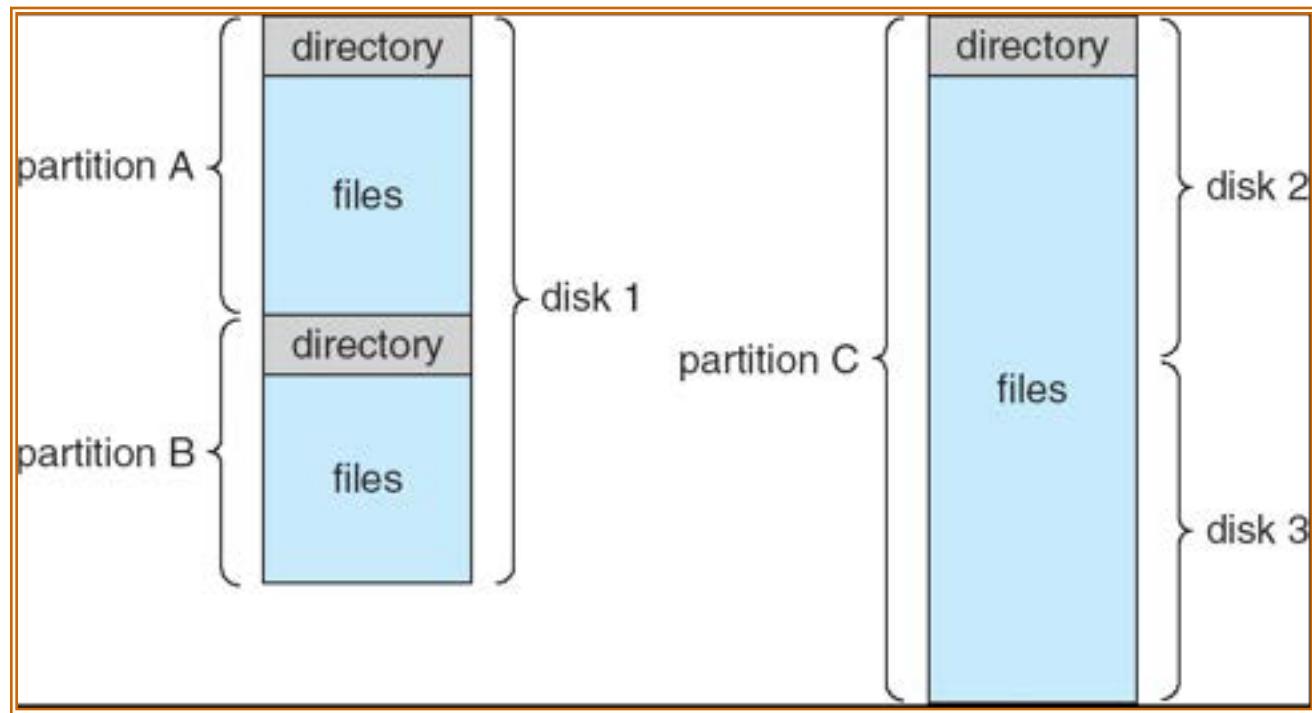
Tipi di directory

Struttura a grafo aciclico (es. UNIX): estende la struttura ad albero con la possibilità di ***inserire link differenti allo stesso file***



Directory e partizioni

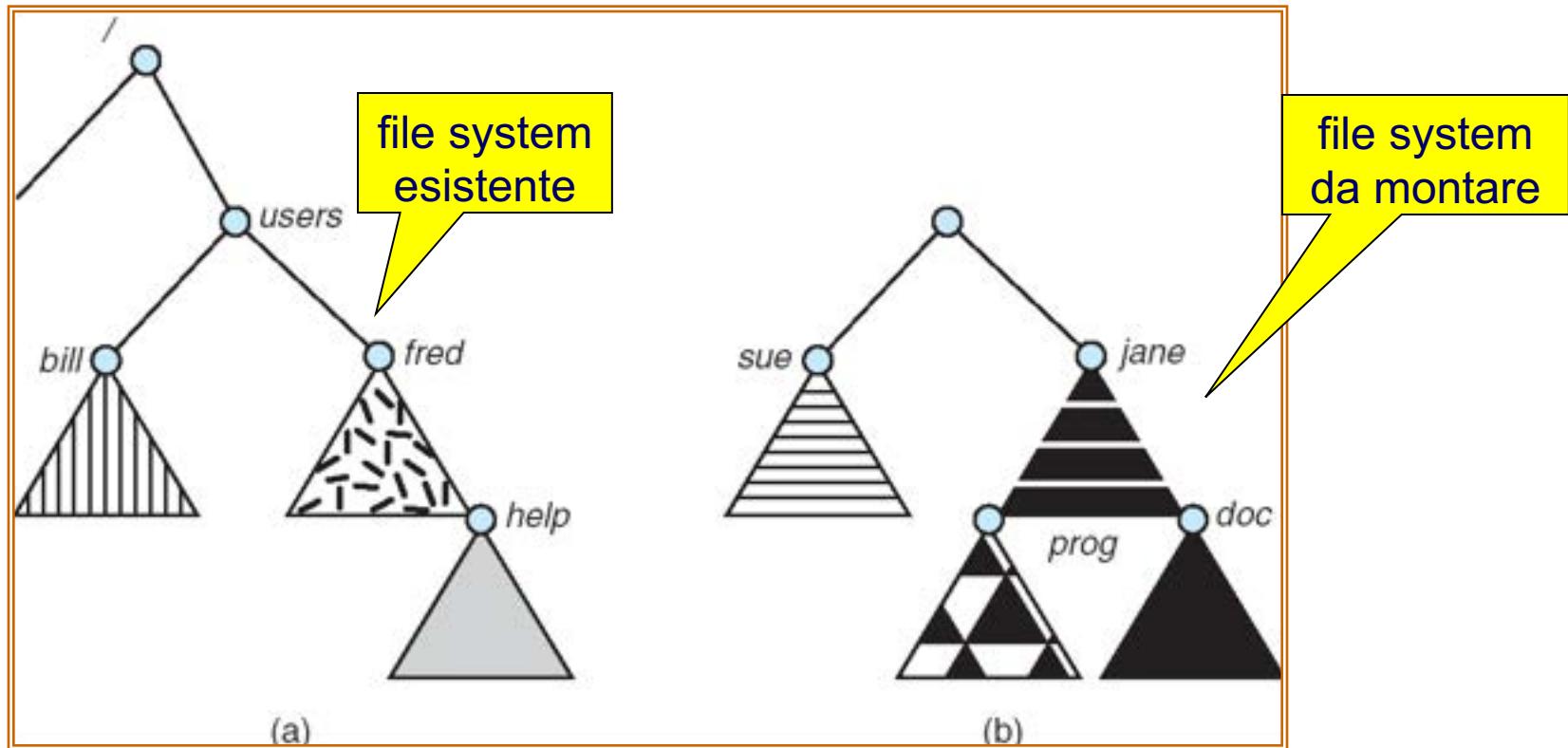
Una singola unità di memorizzazione secondaria (es. Disco) può contenere più partizioni



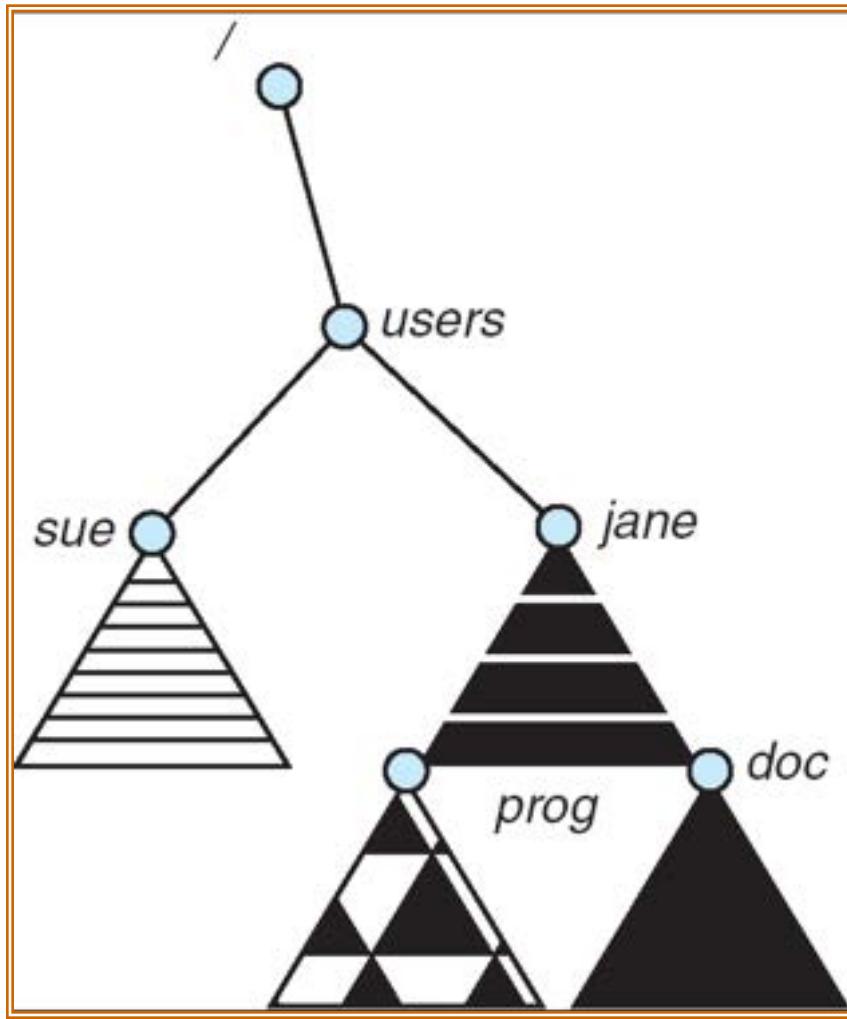
Unità disco e organizzazione/posizione di directory all'interno del file system devono essere correlati?

File System Mounting

Molti SO richiedono il ***mounting esplicito*** all' interno del file system prima di poter usare una (nuova) ***unità disco*** o una ***partizione***



Dopo il mounting ad un determinato mount point



Accesso

Operazioni sui file

Compito del SO è consentire **l'accesso on-line ai file**: ogni volta che un processo **modifica un file**, tale cambiamento è **immediatamente visibile** a tutti gli altri processi.

Tipiche Operazioni

- **Creazione**: allocazione di un file in memoria secondaria e inizializzazione dei suoi attributi
- **Lettura** di record logici dal file
- **Scrittura**: inserimento di nuovi record logici all'interno di file
- **Cancellazione**: eliminazione del file dal file system

- Ogni operazione richiederebbe la localizzazione e l'accesso ad informazioni sulla memoria secondaria (es. disco), come:
- **indirizzi** dei record logici a cui accedere
 - altri attributi del file (**diritti di accesso**, ecc.)
 - **contenuto** del file (record logici)

-> **costo elevato**

Operazioni sui file

Per migliorare l'efficienza:

- SO mantiene **in memoria una struttura che registra i file attualmente in uso (file aperti)** -> **tabella dei file aperti** per ogni file aperto **{puntatore al file, posizione su disco, ...}**
- viene fatto il «**memory mapping**» **dei file aperti**: i file aperti (o porzioni di essi) vengono temporaneamente copiati in memoria centrale → accessi più veloci

Operazioni necessarie

- **Apertura**: introduzione di un **nuovo elemento nella tabella dei file aperti** e eventuale **memory mapping** del file
- **Chiusura**: **salvataggio** del file in memoria secondaria ed **eliminazione** dell'elemento corrispondente dalla **tabella dei file aperti**

Struttura interna dei file

Ogni dispositivo di memorizzazione secondaria viene **partizionato in blocchi (o record fisici)**:

Blocco: unità di **trasferimento fisico** nelle operazioni di I/O da/verso il dispositivo. Sempre di **dimensione fissa**

Le applicazioni vedono il file come un **insieme di record logici**:

Record logico: unità di **trasferimento logico** nelle operazioni di **accesso** al file (es. lettura, scrittura di blocchi). Di **dimensione variabile**

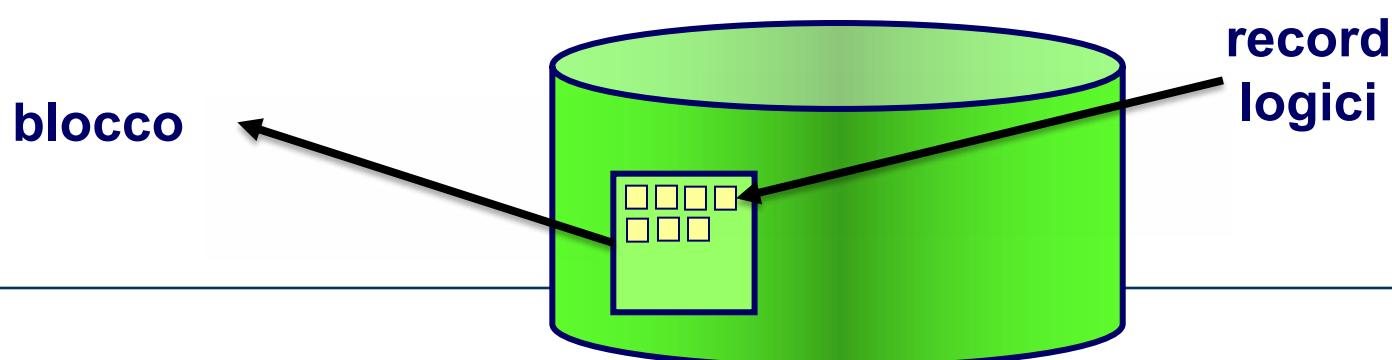
Blocchi & record logici

Uno dei compiti di SO (parte di gestione del file system) è stabilire una ***corrispondenza tra record logici e blocchi***

Di solito:

Dimensione(blocco) >> Dimensione(record logico)

- ***impaccamento*** di record logici all'interno di blocchi



Metodi di accesso

L'accesso a file può avvenire secondo varie modalità:

- **accesso sequenziale**
- **accesso diretto**
- **accesso a indice**

Il metodo di accesso è indipendente:

- . **dal tipo di dispositivo** utilizzato
- . **dalla tecnica di allocazione** dei blocchi in memoria secondaria

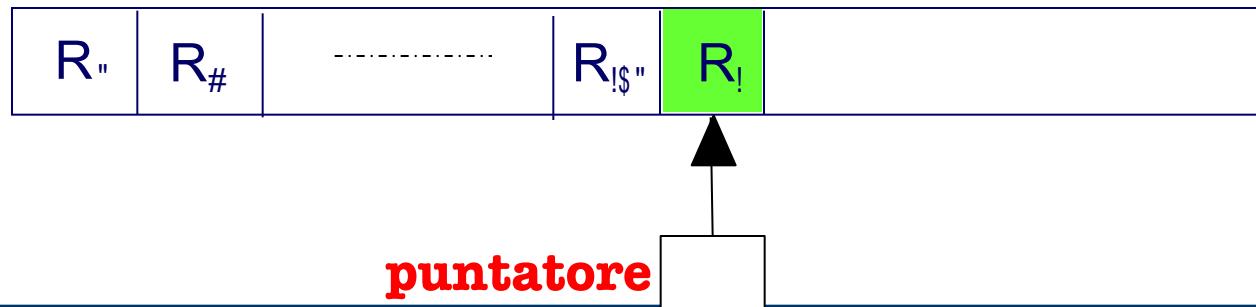
Accesso sequenziale

Il file è una **sequenza** $[R_1, R_2, \dots, R_N]$ di record logici:

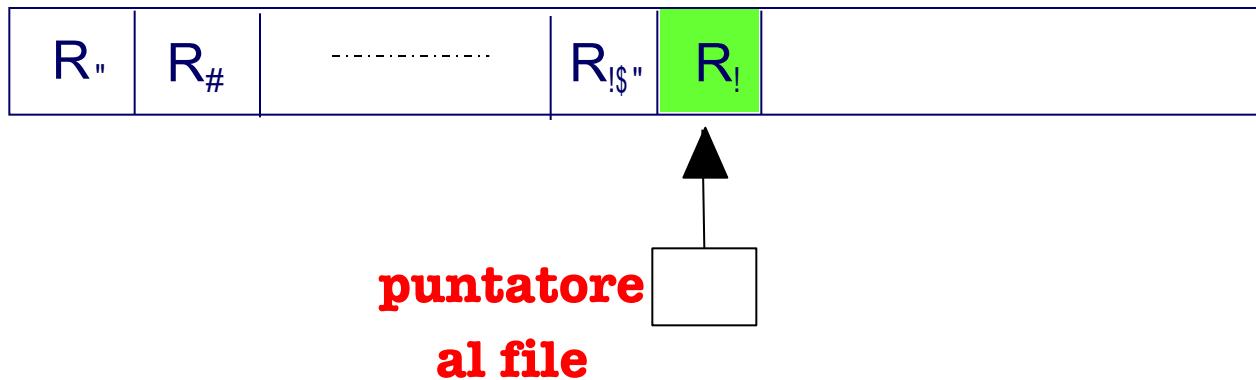
- per accedere ad un particolare record logico R_i , è necessario accedere **prima agli (i-1) record che lo precedono** nella sequenza:



- le operazioni di accesso sono del tipo:
 - ✓ **readnext**: lettura del prossimo record logico della sequenza
 - ✓ **writenext**: scrittura del prossimo record logico
- È necessario registrare la posizione corrente:
puntatore al file (I/O pointer):



Accesso sequenziale



- ogni operazione di accesso (lettura/scrittura) posiziona il **puntatore al file** (current position) sull'elemento successivo a quello letto/scritto

UNIX prevede questo tipo di accesso

Accesso diretto

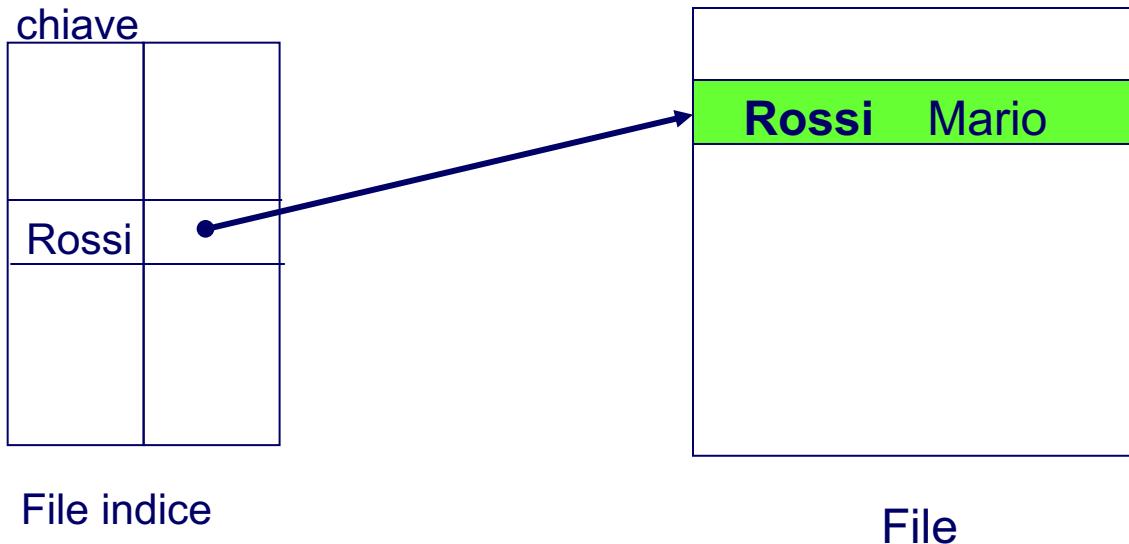
Il file è un **insieme** $\{R_1, R_2, \dots, R_N\}$ di **record logici numerati**:

- si può accedere direttamente a un particolare record logico specificandone il numero
- operazioni di accesso sono del tipo
 - ✓ **read i**: lettura del record logico i
 - ✓ **write i**: scrittura del record logico i
- Utile quando si vuole accedere a **grossi file** per estrarre/aggiornare **poche informazioni** (ad esempio nell'accesso a database)

Accesso a indice

Ad ogni file viene associata una **struttura dati** contenente l'**indice** delle informazioni contenute

- per accedere a un record logico, si esegue **una ricerca nell'indice (utilizzando una chiave)**



File system e protezione

Il **proprietario/creatore** di un file dovrebbe avere la possibilità di **controllare**:

- ❑ quali azioni sono consentite sul file
- ❑ da parte di chi

Possibili azioni su file (“diritti”):

- | | |
|------------------|-----------------|
| - Read | - Write |
| - Execute | - Append |
| - Delete | - List |

Liste di accesso e gruppi (es. UNIX)

Modalità di accesso: read, write, execute

- 3 classi di utenti

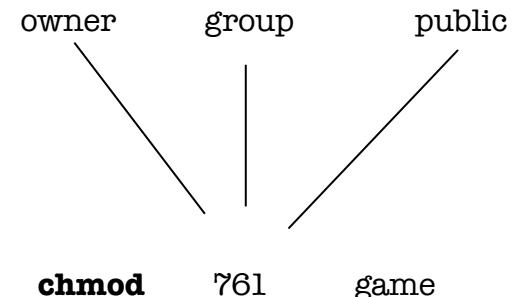
1) owner access	7	RWX
2) group access	6	1 1 0
3) public access	1	0 0 1

- Amministratore può creare gruppi (con nomi unici) e inserire/eliminare utenti in/da quel gruppo
- Dato un file o una directory, si devono definire le regole di accesso desiderate

Cambiamento di gruppo: comando

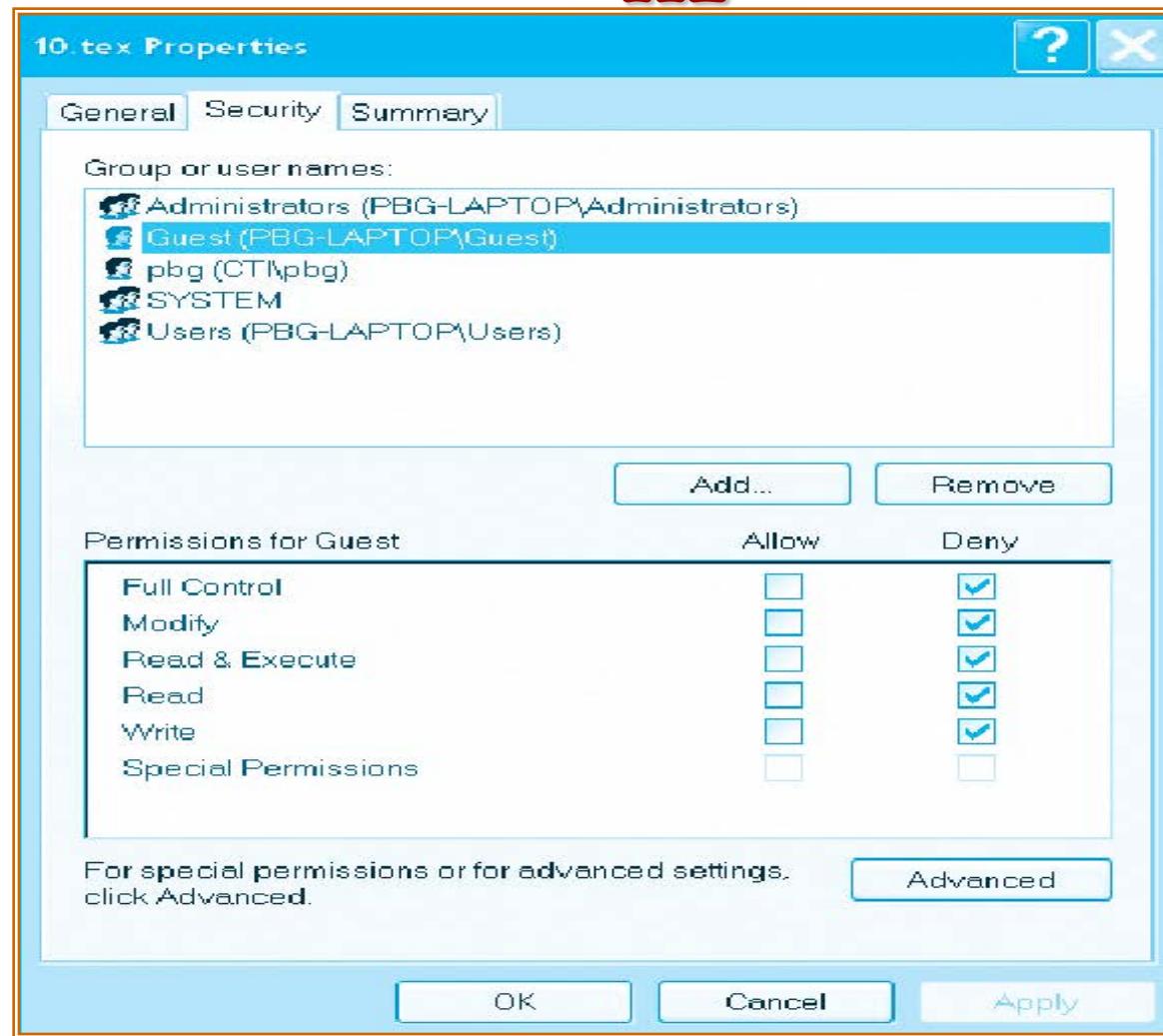
chgrp G game

Cambiamento di diritti di
accesso:



chmod 761 game

Gestione access control list in MS Windows XP



Un esempio di directory listing in UNIX

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

Organizzazione fisica

Organizzazione fisica del file system

SO si occupa anche della **realizzazione del file system sui dispositivi di memorizzazione secondaria**:

- **realizzazione dei descrittori** e loro organizzazione
- **allocazione dei blocchi fisici**
- **gestione dello spazio libero**

Come può essere rappresentato il file system sui dispositivi di memorizzazione secondaria?

Metodi di allocazione

Il **blocco** è l'unità di allocazione sul disco.

Ogni **blocco** contiene un insieme di
record logici contigui

Quali sono le tecniche più comuni per
l'allocazione dei blocchi sul disco?

- allocazione contigua**
- allocazione a lista**
- allocazione a indice**

Allocazione contigua

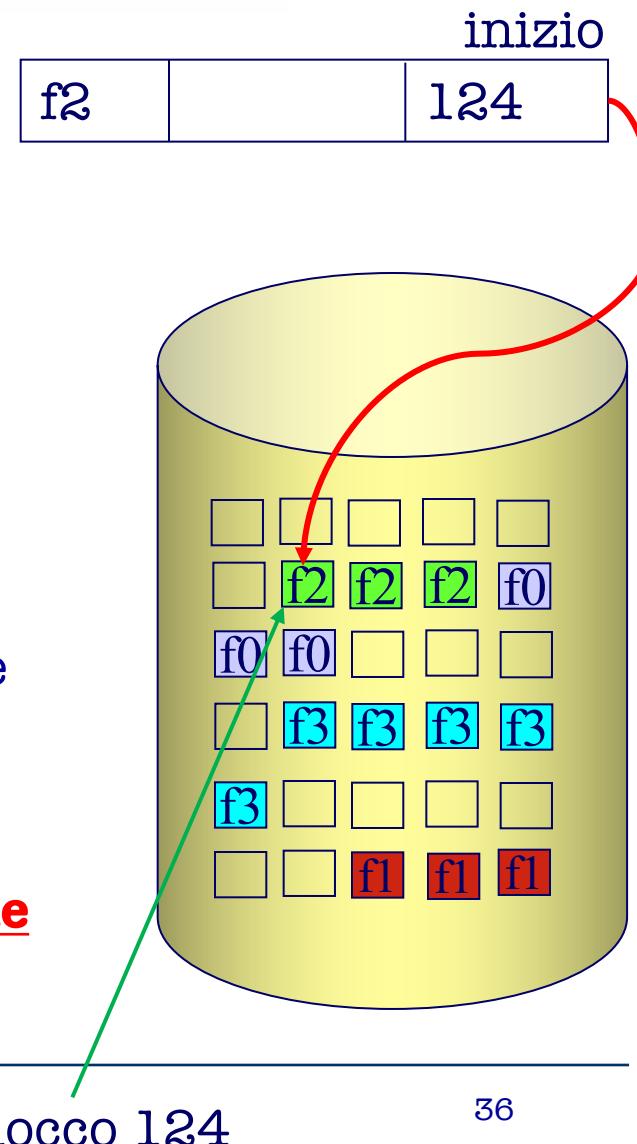
Ogni file è mappato su un insieme di
blocchi fisicamente contigui

Vantaggi

- **costo** della ricerca di un blocco
- possibilità di **accesso sequenziale e diretto**

Svantaggi

- individuazione dello **spazio libero** per l'allocazione di un nuovo file
- **frammentazione esterna**: man mano che si riempie il disco, rimangono zone contigue sempre più piccole, a volte inutilizzabili
 - **Necessità di azioni di compattazione**
- **aumento dinamico delle dimensioni** di file



Allocazione a lista concatenata

I blocchi sui quali viene mappato ogni file sono **organizzati in una lista concatenata**

Vantaggi

- non c'è **frammentazione esterna**
- minor costo di allocazione

Svantaggi:

- possibilità di errore se link danneggiato
- **maggior occupazione** (spazio occupato dai puntatori)
- difficoltà di realizzazione dell'accesso diretto
- **costo della ricerca** di un blocco

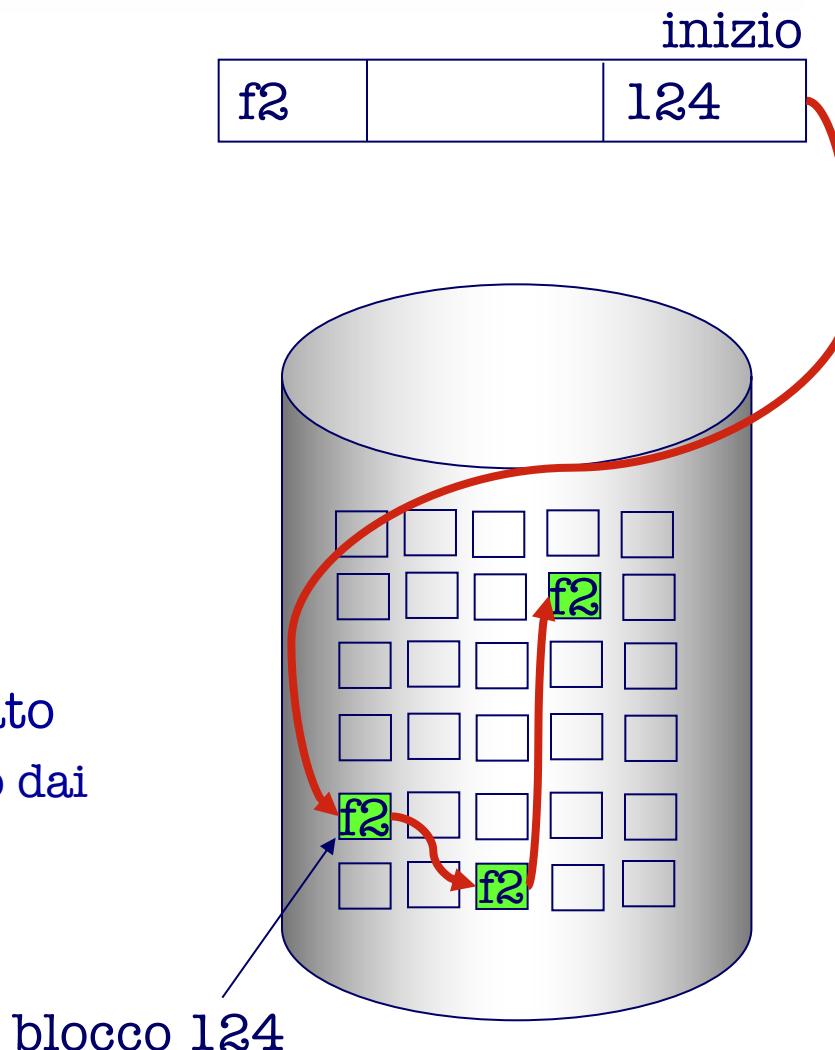


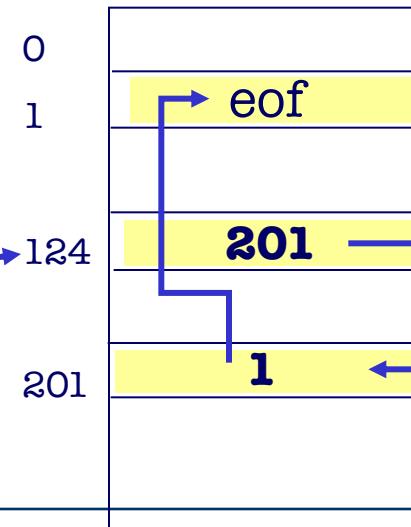
Tabella di allocazione dei file (FAT)

Alcuni SO (ad es. windows, OS/2, dos, ntfs) realizzano **l'allocazione a lista** in modo più efficiente e robusto:

- per ogni partizione, viene mantenuta una tabella (FAT) in cui ogni elemento rappresenta un blocco fisico
- concatenamento dei blocchi sui quali è allocato un file è rappresentato nella FAT

File1		124
-------	--	-----

inizio



Allocazione a indice

Allocazione a lista: i puntatori ai blocchi sono **distribuiti sul disco**

- elevato tempo medio di accesso a un blocco
- complessità della realizzazione del metodo di accesso diretto

Allocazione a indice: tutti i **puntatori ai blocchi** utilizzati per l'allocazione di un determinato file sono **concentrati in un unico blocco per quel file (blocco indice)**

Allocazione a indice

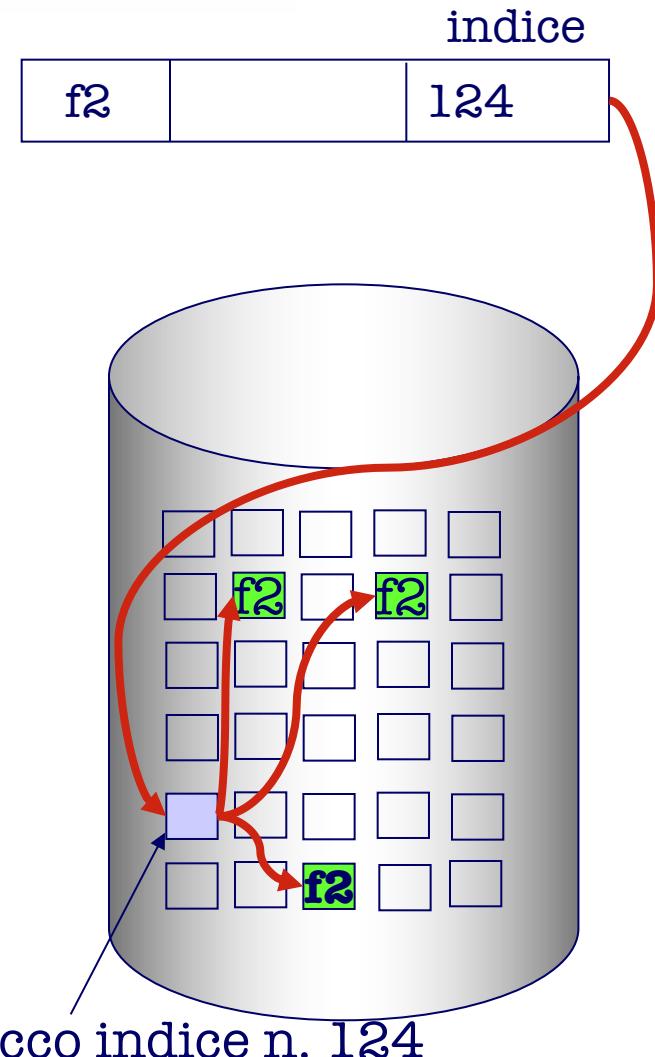
A ogni file è associato un **blocco (indice)** in cui sono contenuti **tutti gli indirizzi dei blocchi** su cui è allocato il file

Vantaggi

- stessi dell'allocazione a lista, più
 - possibilità di accesso diretto
 - maggiore velocità di accesso (rispetto a liste)

Svantaggi

- possibile scarso utilizzo dei blocchi indice



Metodi di allocazione

Riassumendo, gli aspetti caratterizzanti sono:

- grado di **utilizzo della memoria**
- **tempo di accesso medio** al blocco
- realizzazione dei **metodi di accesso**

Esistono SO che adottano **più di un metodo di allocazione**; spesso:

- file **piccoli** → allocazione contigua
- file **grandi** → allocazione a indice

Terza Esercitazione

Gestione di segnali in Unix
Primitive **signal** e **kill**



Primitive fondamentali (sintesi)

signal	<ul style="list-style-type: none">• Imposta la reazione del processo all'eventuale ricezione di un segnale (può essere una funzione handler, SIG_IGN o SIG_DFL)
kill	<ul style="list-style-type: none">• Invio di un segnale ad un processo• Va specificato sia il segnale che il processo destinatario• Restituisce 0 se tutto va bene o -1 in caso di errore• kill -l da shell per una lista dei segnali disponibili
pause	<ul style="list-style-type: none">• Chiamata bloccante: il processo si sospende fino alla ricezione di un qualsiasi segnale
alarm	<ul style="list-style-type: none">• "Schedula" l'invio del segnale SIGALRM al processo chiamante dopo un intervallo di tempo (in secondi) specificato come argomento. Ritorna il numero di secondi mancante allo scadere del time-out precedente.
sleep	<ul style="list-style-type: none">• Sospende il processo chiamante per un numero intero di secondi, oppure fino all'arrivo di un segnale• Restituisce il numero di secondi che sarebbero rimasti da dormire (0 se nessun segnale è arrivato)

Esempio – Segnali di stato e terminazione

- Si realizzi un programma C che utilizzi le primitive Unix per la gestione di processi e segnali, con la seguente interfaccia di invocazione

scopri_terminazione N K

- Il processo iniziale genera **N figli**:

- I primi **K** ($K < N$) processi **attendono** la ricezione del segnale **SIGUSR1** da parte del padre, e poi terminano.

- I **rimanenti** processi **attendono 5 secondi** e poi terminano.

- **Tutti** i figli devono **stampare a video il proprio PID** prima di terminare

Esempio - osservazioni

- Gestire appropriatamente le **attese**:
 - **No attesa attiva (loop)**
 - Quali **primitive** usare per i due tipi di figli?
- Il padre termina K figli tramite **SIGUSR1**
 - Come fa a discriminare a quali figli inviarlo?

Esempio – Soluzione (1/3)

```
int main(int argc, char* argv[]) {
    int i, n, k, pid[MAX_CHILDREN];
    n = atoi(argv[1]);
    k = atoi(argv[2]);
    for(i=0; i<n; i++) {
        pid[i] = fork();
        if (pid[i] == 0) /* Codice Figlio */
            if (i < k)
                wait_for_signal();
            else
                sleep_and_terminate();
        }else if (pid[i] > 0) { /* Codice Padre */
        }else { /* Gestione errori */
    }
    for (i=0; i<k; i++) kill(pid[i], SIGUSR1);
    for (i=0; i<n; i++) wait_child();
    return 0;
}
```

Esempio - Soluzione (2/3)

```
void wait_for_signal(){
    /* Imposto il gestore dei segnali di tipo SIGUSR1 */
    signal(SIGUSR1, sig_usr1_handler);
    pause();
    exit(EXIT_SUCCESS);}

void sig_usr1_handler(int signum) /*Gestione segnale*/
{
    printf("%d: received SIGUSR1(%d). Will
        terminate :-( \n", getpid(), signum);}
```

```
void sleep_and_terminate() {
    sleep(5);
    printf("%d: Slept 5sec. Withdrawing.\n",getpid());
    exit(EXIT_SUCCESS);}
```

```
void wait_child() {
    ... pid = wait(&status);
    /* Gestione condizioni di errore e verifica tipo di
    terminazione (volontaria o da segnale) */
    ...}
```

Esempio - Riflessione A

```
void wait_for_signal(){
    /* Imposto il gestore dei segnali di tipo SIGUSR1 */
    signal(SIGUSR1, sig_usr1_handler);
    pause();
    exit(EXIT_SUCCESS);}

void sig_usr1_handler(int signum) {
    printf("%d: received SIGUSR1(%d)\n"
        "terminate :-( \n", getpid(), signum);}
```

Cosa succede se
SIGUSR1
arrivasse qui!?

```
void sleep_and_terminate() {
    sleep(5);
    printf("%d: Slept 5sec. Withdrawing.\n",getpid());
    exit(EXIT_SUCCESS);}
```

```
void wait_child() {
    ... pid = wait(&status);
    /* Gestione condizioni di errore e verifica tipo di
terminazione (volontaria o da segnale) */
    ...}
```

Esempio – Riflessione A

- Se il segnale **SIGUSR1** inviato dal padre arriva prima che il figlio abbia dichiarato qual è l'handler deputato a riceverlo, (quindi prima di **signal (SIGUSR1, sig_usr1_handler);**), il figlio esegue l'handler di default del segnale **SIGUSR1** : **exit**. Incidentalmente il comportamento è simile a quanto ci era richiesto, ma non verrà eseguita la **printf** di **sig_usr1_handler**.
- Si può evitare con certezza che ciò accada?

Esempio - Riflessione A

Soluzioni possibili:

- Far **dormire** il padre per un po' prima di fargli inviare **SIGUSR1** , ma non ho alcuna certezza che questo risolva sempre il problema!
- Far eseguire la **signal (SIGUSR1, sig_usr1_handler)** al padre prima della creazione dei figli -> il figlio eredita l'associazione segnale-handler. (risolve con certezza il problema, ma va bene solo se il padre non ha bisogno di gestire diversamente SIGUSR1)
- Oppure introdurre una sincronizzazione figli-padre prima dell'invio di **SIGUSR1** :

```
int OKF=0;
int main(int argc, char* argv[]) {
    int i, n, pid[MAX_CHILDREN];
    n = atoi(argv[1]);
    k=atoi(argv[2]);
    signal(SIGUSR2, figlio_ok);
    for(i=0; i<n; i++) {
        pid[i] = fork();
        ...
    }
    while(OKF<k) pause(); //figli pronti
    for (i=0; i<k; i++) kill(pid[i], SIGUSR1);
    for (i=0; i<n; i++) wait_child();
    return 0;
}
..
void wait_for_signal(){
    signal(SIGUSR1, sig_usr1_handler);
    kill(getppid(), SIGUSR2); //figlio pronto
    ...
}
```

```
void figlio_ok(int signum) {
    OKF++;
    printf("figlio %d -simo pronto\n", OKF);
}
```

NB: Questa soluzione risolve con certezza il problema solo in caso di modello affidabile dei segnali, in cui (contrariamente a quanto accade in linux) tutti i segnali ricevuti da un processo sono opportunamente accodati e non vengono mai accorpati

Esempio - Riflessione B

```
void wait_for_signal(){
    /* Imposto il gestore dei segnali di tipo SIGUSR1 */
    signal(SIGUSR1, sig_usr1_handler);
    pause();
    exit(EXIT_SUCCESS);}

void sig_usr1_handler(int signum) /* ... e se SIGUSR1
    printf("%d: received SIGUSR1(%d)
        terminate :-(\ \n", getpid(), signum); */

    arrivasse qui!?
```

```
void sleep_and_terminate() {
    sleep(5);
    printf("%d: Slept 5sec. Withdrawing.\n",getpid());
    exit(EXIT_SUCCESS);}
```

```
void wait_child() {
    ... pid = wait(&status);
    /* Gestione condizioni di errore e verifica tipo di
terminazione (volontaria o da segnale) */
    ...
}
```

Esempio – Riflessione B

- Se il segnale **SIGUSR1** arriva dopo la dichiarazione dell'handler, ma prima della **pause()**?
- Il figlio riceve il segnale, esegue correttamente l'handler e si mette in attesa... di un segnale che è già arrivato!
=> il figlio attende all'infinito!
- Si può evitare tutto ciò? **SI!**
- Mettendo nell' handler **TUTTE** le operazioni che il figlio deve fare alla ricezione del segnale, **inclusa la exit** :

```
void sig_usr1_handler(int signum){  
    printf("%d: received SIGUSR1 (%d) . I was  
    rejected :-( \n", getpid(), signum);  
    exit(EXIT_SUCCESS);  
}
```

I Thread in Java

parte 1

Thread

Un **thread** è un **singolo flusso sequenziale** di controllo all'interno di un processo (**task**).

Un thread (o processo leggero) è un'unità di esecuzione che **condivide codice e dati** con altri thread ad esso associati

Un **thread**

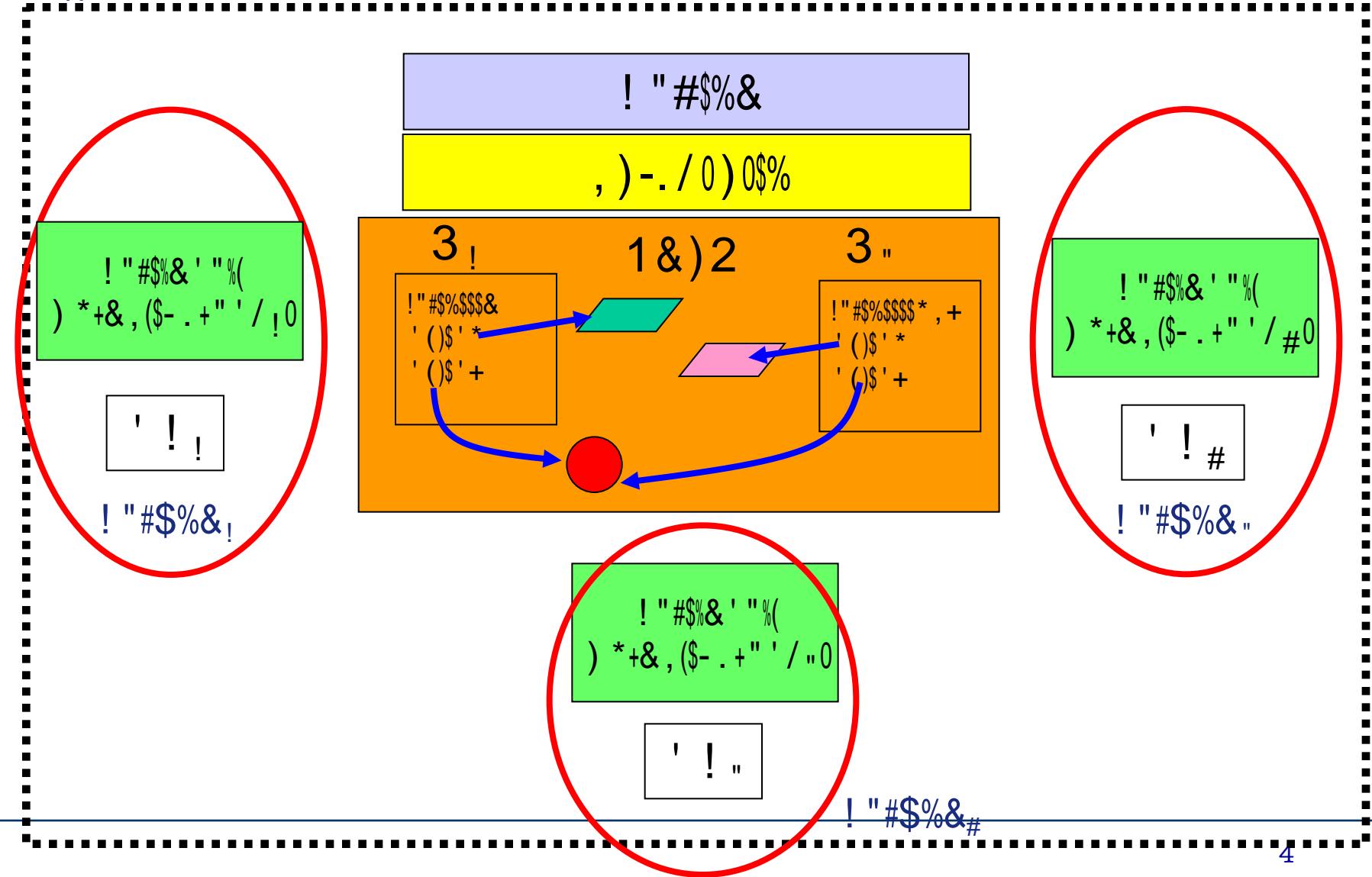
- **NON** ha spazio di memoria riservato per dati e heap: tutti i thread appartenenti allo stesso processo condividono lo **stesso spazio di indirizzamento**
- ha **stack** e **program counter privati**

I threads in Java

- All'esecuzione di ogni programma Java corrisponde un task che contiene almeno un singolo thread, corrispondente all'esecuzione del metodo **main()** sulla JVM.
- E' possibile creare dinamicamente nuovi thread attivando concorrentemente le loro esecuzioni all'interno del programma.

Java Thread

() * +



Java Thread: programmazione

Due modalità per implementare i thread
in Java:

- 1. estendendo la classe Thread**
- 2. implementando l'interfaccia Runnable**

1. Thread come oggetti di sottoclassi della classe Thread

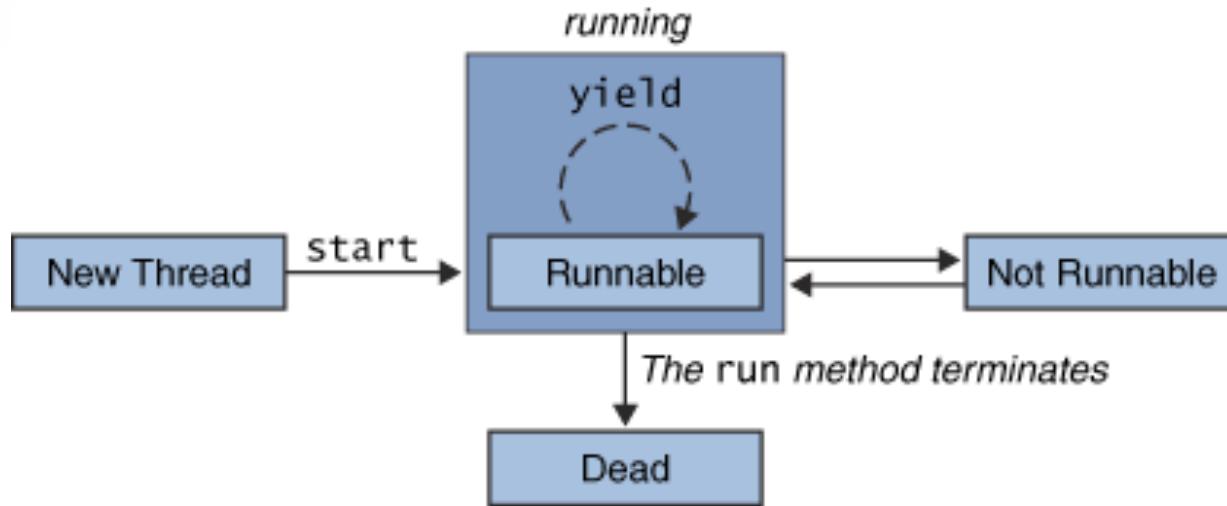
- I thread sono oggetti che derivano dalla **classe Thread** (fornita dal package **java.lang**).
- Il metodo **run()** della classe di libreria **Thread** definisce l'insieme di istruzioni Java che ogni thread (oggetto della classe) eseguirà. (NB: nella classe **Thread** l'implementazione del metodo **run** è vuota).
- In ogni sottoclasse derivata da **Thread** il metodo **run** deve essere **ridefinito (override)** specificando all'interno di esso cosa far eseguire ai thread di quella classe.
- Per creare un thread, si deve creare (tramite **new**) un'istanza della classe che lo definisce; dopo la new il thread esiste, ma **non è ancora attivo**.
- Per attivare un thread si deve invocare il metodo **start()** (che a sua volta invocherà il metodo **run()**).

Possible schema

```
class SimpleThread extends Thread {  
    public void SimpleThread()  
    { <costruttore> }  
  
    public void run() {  
        <sequenza di istruzioni eseguita>  
        <da ogni thread di questa classe>  
    }  
}  
  
public class EsempioConDueThreads  
{  
    public static void main (string[] args)  
    {  
        SimpleThread t1=new SimpleThread();  
        t1.start(); //attivazione del thread t1  
        <resto del programma eseguito  
         dal thread main>  
    }  
}
```

- La classe **SimpleThread** (estensione di Thread) implementa i nuovi thread ridefinendo il metodo **run**.
 - La classe EsempioConDueThreads fornisce il **main** nel quale viene creato il thread **t1** come oggetto derivato dalla classe **Thread**.
 - Per **attivare** il thread deve essere chiamato il metodo **start()** che invoca il metodo **run()** (il metodo **run()** non può essere chiamato direttamente, ma solo attraverso **start()**).
- Abbiamo creato due thread concorrenti: il thread principale associato al **main** ed il thread **t1**.

Ciclo di vita di un thread



New Thread

- Subito dopo l' istruzione **new**
- Il costruttore alloca e inizializza le variabili di istanza

Runnable

- Il thread è eseguibile ma potrebbe non essere in esecuzione

Ciclo di vita di un thread

Not Runnable

- ❑ Il thread non può essere messo in esecuzione perché è in attesa di un evento
- ❑ Entra in questo stato, ad esempio, quando è in attesa della terminazione di un' operazione di I/O, oppure è bloccato a causa di **sincronizzazione** (es. cerca di eseguire una sezione critica su un oggetto occupato, o dopo aver invocato uno dei seguenti metodi: sleep(), wait(), suspend())
- ❑ Esce da questo stato quando si verifica la condizione complementare

Dead

- ❑ Il thread giunge a questo stato per “morte naturale” o perché un altro thread ha invocato il suo metodo stop()

Esempio: primo metodo

```
public class SimpleThread extends Thread{  
  
    public SimpleThread(String str)  
    {super(str);}  
  
    public void run() {  
        for(int i=0; i<10; i++)  
        { System.out.println(i+ " " +getName());  
        try{  
            sleep((int)Math.random()*1000);  
        } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! "+getName());  
    }  
}
```

Java Thread

```
public class EsempioConDueThreads
{
    public static void main(String[] args)
    { SimpleThread st1= new SimpleThread("Pippo");
      st1.start();
    }
}
```

**E se occorre definire thread che
non siano necessariamente
sottoclassi di Thread?**

2. Thread come classi che implementano Runnable

Definizione di thread come classe che implementa l'**interfaccia Runnable**:

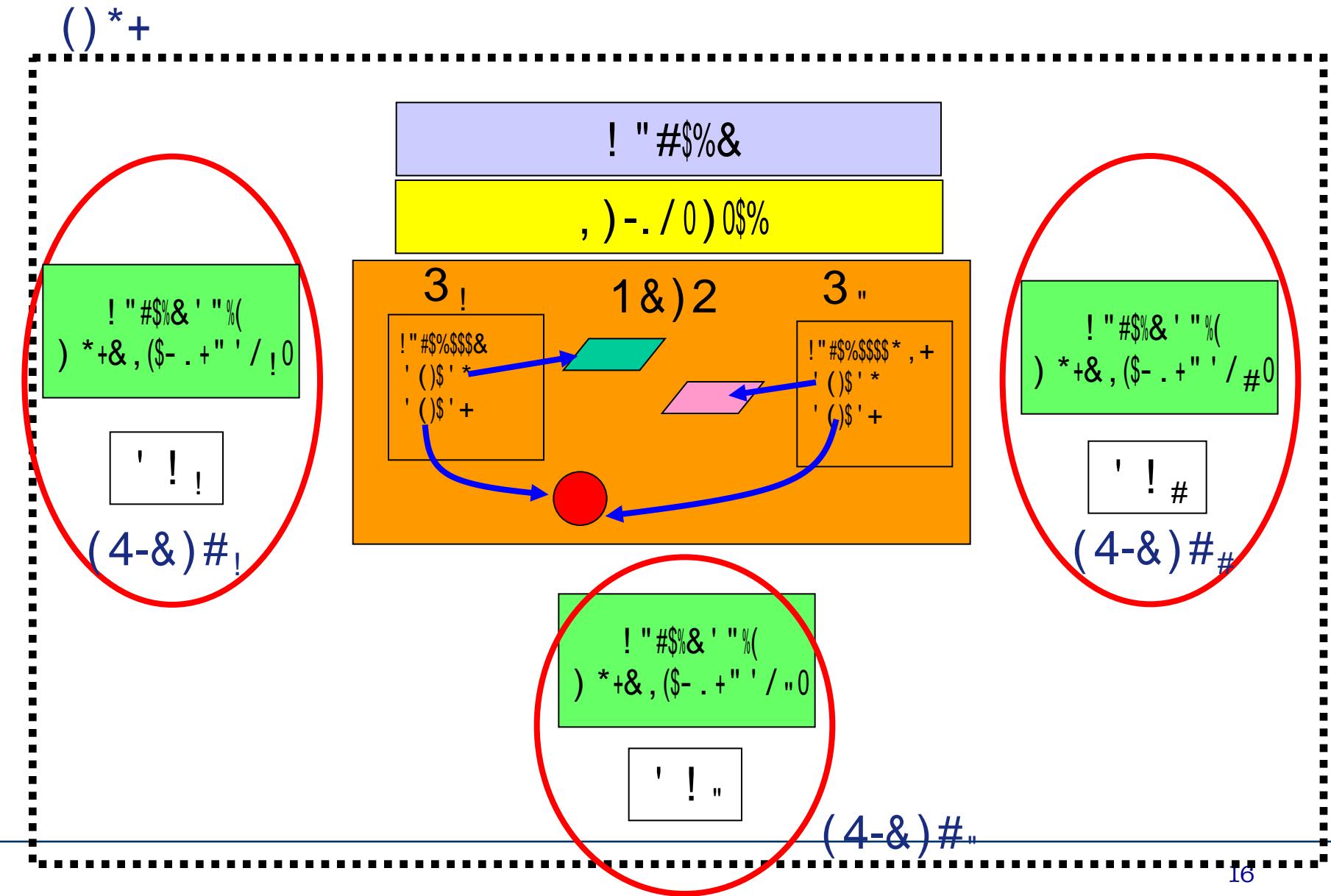
1. la classe deve **ridefinire il metodo run ()**
2. si crea un' istanza di tale classe tramite **new**
3. si crea **un'istanza della classe Thread** con **new**, passandole come **parametro l'oggetto che implementa Runnable**
4. si esegue il thread invocando il metodo **start () sull'oggetto con classe Thread creato**

Esempio: secondo metodo

```
class EsempioRunnable extends MiaClasse
    implements Runnable {
public void run() {
    for (int i=1; i<=10; i++)
        System.out.println(i + " " + i*i);
}
}

public class Esempio {
    public static void main(String args[]) {
        EsempioRunnable e = new EsempioRunnable();
        Thread t = new Thread (e);
        t.start();
    }
}
```

Thread



Sincronizzazione di thread

Differenti thread **condividono lo stesso spazio di memoria (heap):**

- è possibile che più thread accedano **contemporaneamente** a uno stesso oggetto, **invocando un metodo che modifica lo stato dell' oggetto**
- lo stato **finale** dell' oggetto dipenderà **dall' ordine** con cui i thread accedono ad esso.

→ Servono meccanismi di **sincronizzazione**

Sincronizzazione in Java

Siamo nel modello ad ambiente globale (o a memoria comune):

→ Ogni tipo di interazione tra thread avviene tramite **oggetti comuni**:

- Interazione di tipo **competitivo (mutua esclusione)**:
 - meccanismo degli **objects locks**
 - blocchi e metodi **synchronized**
- Interazione di tipo **cooperativo**:
 - meccanismo **wait-notify -> semafori**
 - **[variabili condizione]**

Mutua esclusione

- Ad ogni oggetto viene associato un oggetto «**object lock**» che rappresenta lo stato dell'oggetto (libero/occupato).
- L'associazione del lock ad ogni oggetto viene fatta in modo **automatico** dalla JVM.
- E' possibile denotare alcune sezioni di codice che operano su un oggetto come **sezioni critiche** tramite la parola chiave **synchronized**.

→ Per ogni parte di codice **synchronized** compilatore inserisce:

- un prologo in testa alla sezione critica per **l'acquisizione dell'object lock** associato all'oggetto (v. lock()).
- un epilogo alla fine della sezione critica per **rilasciare l'object lock** (v. unlock()).

Blocchi synchronized

Con riferimento ad un oggetto x, si può definire un blocco di statement come una sezione critica nel seguente modo (**synchronized blocks**):

```
synchronized (oggetto x) {<sequenza di statement>;}
```

Esempio:

```
Object mutexLock= new Object;  
....  
public void M( ) {  
    <sezione di codice non critica>;  
    synchronized (mutexlock){  
        < sezione di codice critica>;  
    }  
    <sezione di codice non critica>;  
}
```

- all'oggetto **mutexLock** viene implicitamente associato un lock, il cui valore puo` essere:
 - **libero**: il thread può eseguire la sezione critica
 - **occupato**: nel tentativo di eseguire la sezione critica, il thread viene sospeso dalla JVM in una coda associata a **mutexLock** (**entry set**).

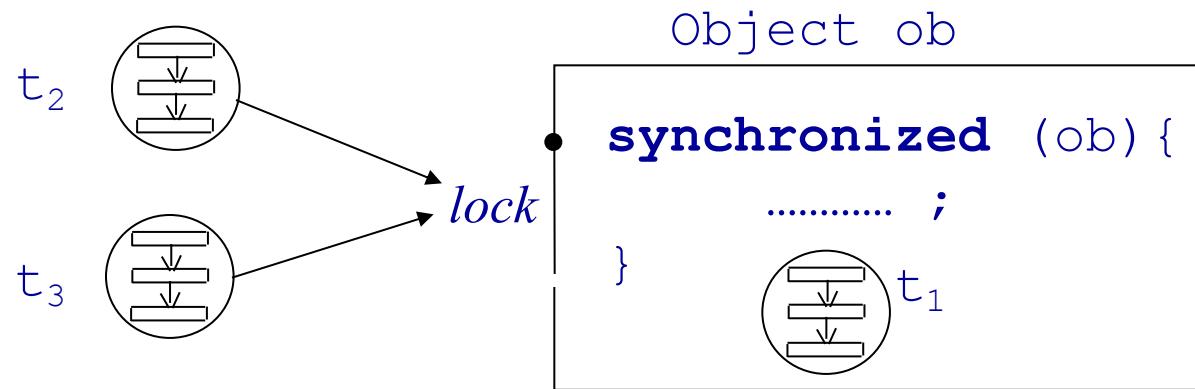
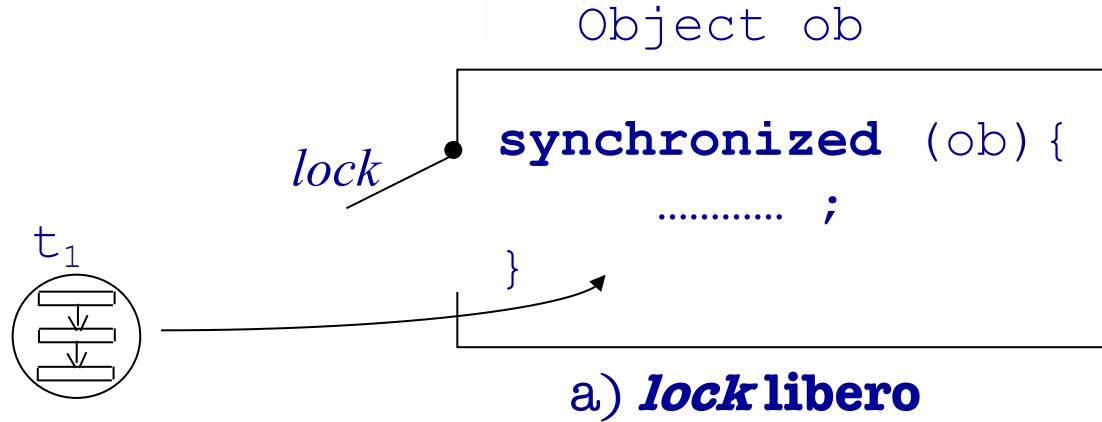
Al termine della sezione critica:

- se non ci sono thread in attesa: il lock viene reso libero .
- se ci sono thread in attesa: il lock rimane occupato e viene riattivato uno di questi .

synchronized block

- esecuzione del blocco **mutuamente esclusiva** rispetto:
 - ad altre esecuzioni dello stesso blocco
 - all'esecuzione di altri blocchi sincronizzati sullo stesso oggetto

Entry set di un oggetto



Metodi synchronized

- **Mutua esclusione** tra i metodi di una classe

```
public class contatore {  
    private int i=0;  
    public synchronized void incrementa()  
    { i ++; }  
    public synchronized void decrementa()  
    {i--; }  
}
```

- Quando un metodo viene invocato per operare su un oggetto della classe, l'esecuzione del metodo avviene in **mutua esclusione utilizzando il lock dell'oggetto.**

Esempio: accesso concorrente a un contatore

```
public class competingproc extends Thread
{ contatore r; /* risorsa condivisa */
  int T; // incrementa se tipo=1; decrementa se tipo=-1

  public competingproc(contatore R,  int tipo)
  {   this.r = R;
      this.T = tipo;
  }

  public void run()
  {   try{
      while(true)
      {   if (T>0)          r.incrementa();
          else if (T<0)    r.decrementa();
      }
      }catch(InterruptedException e) {}
  }
}
```

```
public class contatore {  
    private int C;  
  
    public contatore(int i)  
    { this.C=i; }  
  
    public synchronized void incrementa()  
    { C++;  
        System.out.print("\n eseguito incremento: valore  
attuale del contatore: "+ C+" ....\n");  
    }  
  
    public synchronized void decrementa()  
    { C--;  
        System.out.print("\n eseguito decremento: valore  
attuale del contatore: "+ C+" ....\n");  
    }  
}
```

```
import java.util.*;  
  
public class prova_mutex{ // test  
  
    public static void main(String args[]) {  
        final int NP=30;  
        contatore C =new contatore(0);  
        competingproc []F=new competingproc[NP];  
        int i;  
        for(i=0;i<(NP/2);i++)  
            F[i]=new competingproc(C, 1); // incrementa  
        for(i=(NP/2);i<NP;i++)  
            F[i]=new competingproc(C, -1); // decrementa  
        for(i=0;i<NP;i++)  
            F[i].start();  
    }  
}
```

Semafori in Java

Dalla versione 5.0, in Java è disponibile la classe **Semaphore**:

```
import java.util.concurrent.Semaphore;
```

Tramite la quale si possono creare semafori, sui quali è possibile operare tramite i metodi:

- **acquire();** // implementazione di p()
- **release();** // implementazione di v()

Uso di oggetti Semaphore:

Inizializzazione ad un valore K dato:

```
Semaphore s=new Semaphore(k);
```

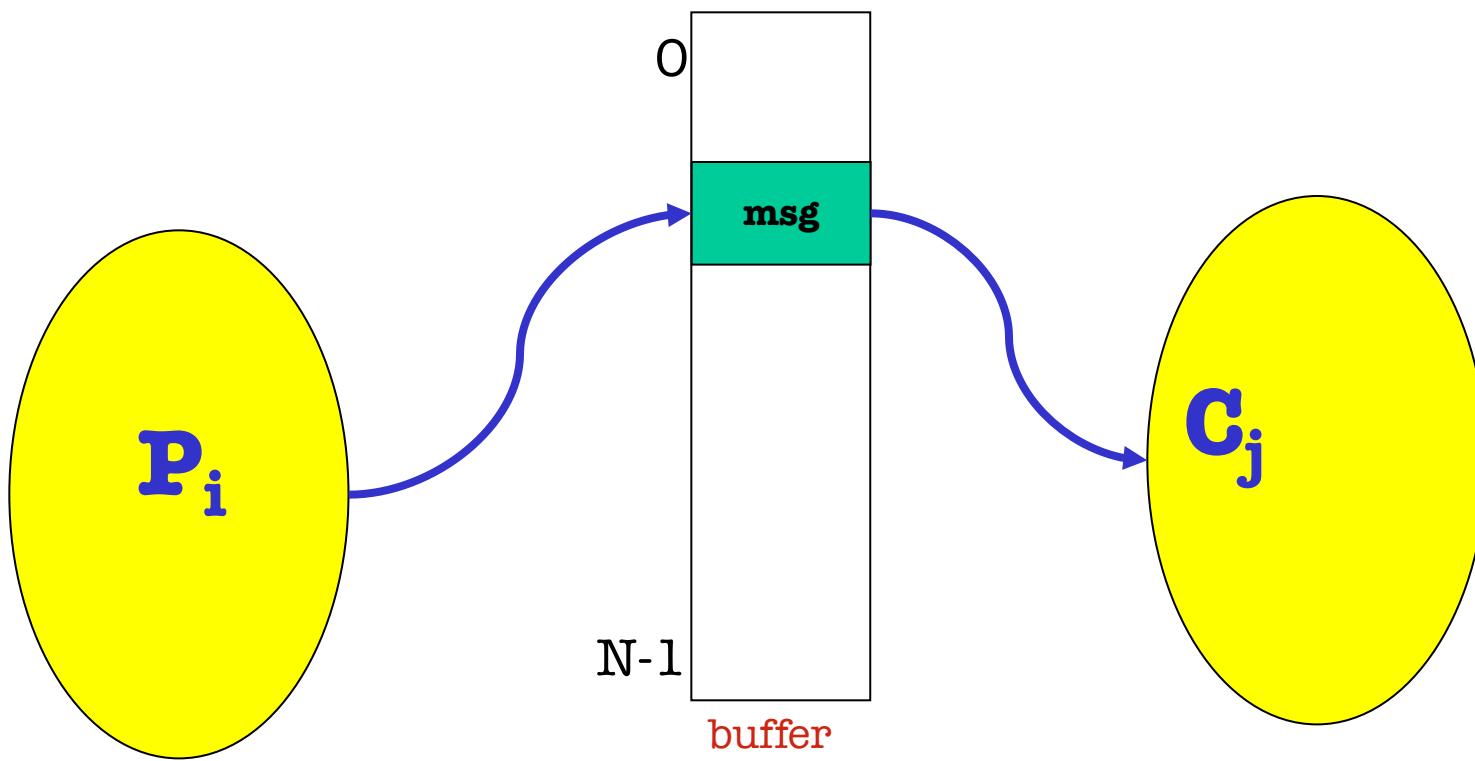
Operazioni: stessa semantica di p e v

```
s.acquire(); // esecuzione di p() su s
```

```
s.release(); // esecuzione di v() su s
```

NB Esistono altre operazioni che estendono la semantica tradizionale del semaforo. (<http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Semaphore.html>)

Esempio: produttori e consumatori con buffer di capacità N



HP: Buffer (mailbox) limitato di dimensione N

```
public class threadP extends Thread{ //produttori
    int i=0;
    risorsa r; //oggetto che rappresenta il buffer condiviso

    public threadP(risorsa R)
    {   this.r=R;
    }

    public void run()
    {   try{   System.out.print("\nThread PRODUTTORE: il mio
        ID è: "+getName()+"..\n");
        while (i<100)
        { sleep(100);
            r.inserimento(i);
            i++;
            System.out.print("\n"+ getName() +":"
                inserito messaggio " +i+ "\n");
        }
    }catch(InterruptedException e) {}
}
}
```

```
public class threadC extends Thread{ //consumatori
    int msg;
    risorsa r;

    public threadC(rorsa R)
    {   this.r=R;
    }

    public void run()
    {   try{   System.out.print("\nThread CONSUMATORE: il mio
        ID è: "+getName()+"..\n");
        while (true)
        {       msg=r.prelievo();
                System.out.print("\n"+getName()+""
                consumatore ha letto il messaggio "+msg
                + "...\\n");
            }
    }catch(InterruptedException e) {}
}
}
```

```
import java.util.concurrent.Semaphore;
public class risorsa {
// definizione buffer condiviso:
    final int N = 30;      // capacità del buffer
    int lettura, scrittura;//indice di lettura
    int []buffer;
    Semaphore sP; /* sospensione dei Produttori; v.i. N*/
    Semaphore sC; /* sospensione dei Consumatori v.i. 0*/
    Semaphore sM; // semaforo di mutua esclusione v.i. 1

    public risorsa() // costruttore
    {   lettura=0;
        scrittura=0;
        buffer= new int[N];
        sP=new Semaphore(N); /* v.i. N*/
        sC=new Semaphore(0); /* v.i. 0*/
        sM=new Semaphore(1); // semaforo di mutua esclusione
    } //continua..
```

```
// ..continua classe risorsa
```

```
public void inserimento(int M)
{ try{ sP.acquire();
    sM.acquire(); //inizio sez critica
    buffer[scrittura]=M;
    scrittura=(scrittura+1)%N;
    sM.release(); //fine sez critica
    sC.release();
}catch(InterruptedException e) {}
}
```

```
//continua..
```

//... Continua

```
public int prelievo()
{   int messaggio=-1;
    try{ sC.acquire();
          sM.acquire(); //inizio sez critica
          messaggio=buffer[lettura];
          lettura=(lettura+1)%N;
          sM.release(); //fine sez critica
          sP.release();
      }catch(InterruptedException e){ }
    return messaggio;
}
} // fine classe risorsa
```

```
import java.util.concurrent.*;  
  
public class prodcons{  
  
    public static void main(String args[]) {  
  
        risorsa R = new risorsa(); // creaz. buffer  
        threadP TP=new threadP(R);  
        threadC TC=new threadC(R);  
  
        TC.start();  
        TP.start();  
    }  
}
```

Esempio: la Catena di Montaggio

Un'azienda elettronica produce schede a microprocessore. La produzione di ogni scheda avviene in due fasi diverse:

- 1) **Assemblaggio**: in questa fase avviene l'assemblaggio automatico dei diversi componenti;
- 2) **Inscatolamento**: le schede assemblate vengono introdotte in scatole di capacità N.

Si supponga di affidare **ognuna delle 2 fasi a un thread distinto** incaricato di controllare la macchina automatica dedicata alla realizzazione di quella particolare fase.

Utilizzando i **semafori**, si realizzi una politica di sincronizzazione che tenga conto dei seguenti vincoli:

- l'inscatolamento può essere attivato soltanto quando N nuovi prodotti sono stati assemblati.

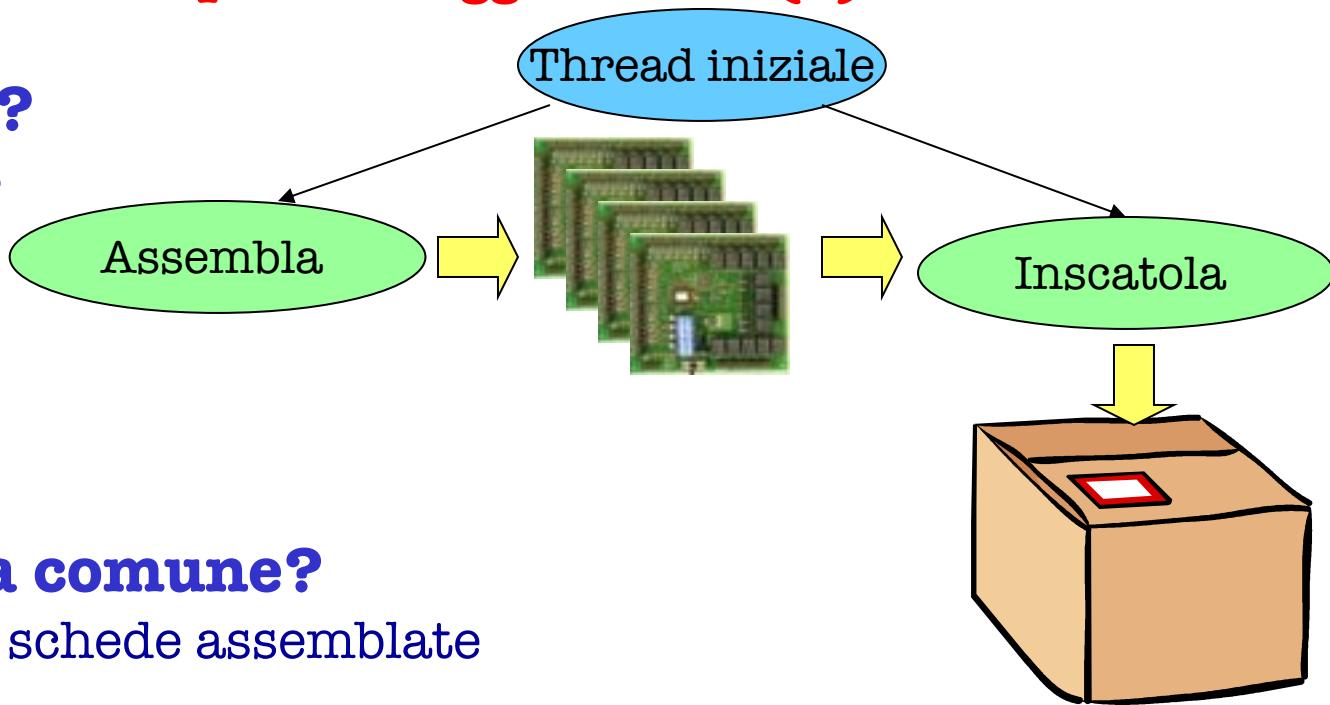
Si supponga inoltre che il thread dedicato all'assemblaggio non possa effettuare una nuova fase di assemblaggio se vi sono ancora **MAX** schede da inscatolare ($MAX > N$).

La soluzione dovrà consentire un soddisfacente grado di concorrenza tra i threads.

Spunti e suggerimenti (1)

Quali thread?

- thread iniziale
- **Assembla**
- **Inscatola**



Quale risorsa comune?

- l'insieme delle schede assemblate

Il thread iniziale crea i 2 thread **Assembla**, **Inscatola**, ognuno con struttura ciclica:

- **Assembla**: al termine di un assemblaggio, soltanto dopo aver prodotto l' N -esima scheda, il thread Assembla attiva il thread *Inscatolamento*; Assembla deve sospendersi se ci sono MAX schede ancora da inscatolare.
- **Inscatola**: una volta attivato provvede ad eseguire la fase di inscatolamento.

Spunti e suggerimenti (2)

Sincronizzazione:

- **mutua esclusione nell'accesso alle risorse condivise (assemblati):**
 - definiamo un **semaforo di mutua esclusione sM**
- **sospensione del processo che Assembلا:**
 - definiamo un **semaforo sA (v.i.=max)**
- **sospensione del processo che Inscatola:**
 - definiamo un **semaforo sI (v.i.=0)**

```
public class threadA extends Thread{ // def. Assemblatore
    int i=0;
    int da_produrre;
    risorsa r;

    public threadA(rorsa R, int pezzi)
    {   this.r=R;
        this.da_produrre=pezzi;
    }

    public void run()
    {   try{
        System.out.print("\nThread ASSEMBLAGGIO: il mio ID
                        è: "+getName()+"..\n");
        while (i<30)
            {
                sleep(100);
                r.nuovoA();
                i++;
                System.out.print("\n"+ getName() +":
                                nuovo assemblato...." +i+ "\n");
            }
    }catch(InterruptedException e) {}

}
```

```
public class threadS extends Thread{ //thread che inscatola  
int i, scatole=0;  
risorsa r;  
  
public threadS(risorsa R)  
{ this.r=R; }  
  
public void run()  
{ try{ System.out.print("\nThread INSCATOLAMENTO:  
il mio ID è: "+getName()+"..\n");  
while (true)  
{ r.nuovaS();  
sleep(100); /*durata inscatolamento...*/  
scatole++;  
System.out.print("\n"+getName()+"  
inscatolamento "+scatole + ... \n");  
}  
}catch(InterruptedException e){}  
} //chiude run  
}
```

```
import java.util.concurrent.Semaphore;

public class risorsa {
    final int max=15;
    final int N = 4; // capacità della scatola
    int pronti; // assemblati pronti (al massimo MAX)

    int i;
    Semaphore sA; // per la sospensione di TA; v.i max
    Semaphore sI; // v.i. 0 per la sospensione di TS
    Semaphore sM; // semaforo di mutua esclusione

    public risorsa(int pronti)
    {   sA=new Semaphore(max); //rappresenta lo spazio
        //disponibile
        sI=new Semaphore (0); // rappresenta il numero di
        //scatole che possono essere
        //confezionate
        sM=new Semaphore (1); // mutua esclusione
        pronti =0;
    }
}
```

```

public void nuovoA() //deposito assemblato
{   try{   sA.acquire();
          sM.acquire(); //inizio sez critica
          pronti=pronti+1;
          if (pronti%N==0)
              sI.release();
          sM.release(); //fine sez critica
      }catch(InterruptedException e) {}
}

public void nuovaS() //preleva N assemblati
{   try{   sI.acquire();
          sM.acquire();
          pronti=pronti-N;
          for(i=0; i<N; i++)
              sA. release();
          sM.release();
      }catch(InterruptedException e) {}
}

}

```

```
public class supplychain{  
  
    public static void main(String args[]) {  
  
        risorsa R=new risorsa(0);  
        threadA TA=new threadA(R, 1000);  
        threadS TS=new threadS(R);  
  
        TA.start();  
        TS.start();  
  
    }  
  
}
```

Java Thread: Alcune considerazioni al contorno: i problemi di stop() e suspend()

stop()

- forza la **terminazione** di un thread
- tutte le **risorse utilizzate vengono immediatamente liberate** (lock compresi)

Se il **thread interrotto** stava compiendo un insieme di operazioni da eseguirsi in maniera **atomica**, l'interruzione può condurre ad uno **stato inconsistente del sistema**.

Per questo motivo il metodo stop() è “deprecated”.

I problemi di `stop()` e `suspend()`

`suspend()`

- **blocca l'esecuzione di un thread**, in attesa di una successiva invocazione di `resume()`
- non libera le risorse impegnate dal thread (**non rilascia i lock**)

Se il **thread sospeso** aveva acquisito una **risorsa** in maniera **esclusiva** (ad esempio sospeso durante l'esecuzione di un metodo synchronized), tale **risorsa rimane bloccata**.

Per questo motivo il metodo suspend() è “deprecated”.

Altri metodi di interesse per Java thread

- **sleep(long ms)**
 - sospende thread per il # di ms specificato
- **join()**
 - **attende la terminazione del thread specificato**
- **interrupt()**
 - invia un evento che produce l' interruzione di un thread
- **interrupted() / isInterrupted()**
 - verificano se il thread corrente è stato interrotto
- **isAlive()**
 - true se thread è stato avviato e non è ancora terminato
- **yield()**
 - costringe il thread a cedere il controllo della CPU

Il monitor

Il costrutto monitor [Hoare '74]

Definizione: Costrutto **sintattico** che associa un insieme di operazioni (**entry**, o *public*) ad una struttura dati comune a più processi, tale che:

- Le operazioni **entry siano le sole operazioni permesse** su quella struttura.
- Le operazioni **entry siano mutuamente esclusive**: un solo processo per volta può essere attivo nel monitor.

Struttura del Monitor (*pseudocodice*)

```
monitor tipo_risorsa {
    <dichiarazioni variabili locali>;
    <inizializzazione variabili locali>

    entry void op1 ( ) {
        <corpo della operazione op1 >;
    }

    ...
    entry void opn ( ) {
        <corpo della operazione opn>;
    }

    <eventuali operazioni non entry>
}
```

- Le variabili locali sono **accessibili solo all'interno del monitor**.
- Le **operazioni entry (o public)** sono **le sole operazioni** che possono essere utilizzate dai processi per accedere alle variabili locali. L'accesso avviene in **modo mutuamente esclusivo**.
- Le **variabili locali** mantengono il loro valore tra successive esecuzioni delle operazioni del monitor (variabili permanenti).
- Le operazioni **non** dichiarate **entry** non sono accessibili dall'esterno. Sono invocabili solo all'interno del monitor (dalle funzioni entry e da quelle non entry).

Esempio (*pseudocodice*)

```
tipo_risorsa ris;
```

- crea una istanza del monitor, cioè una struttura dati organizzata secondo quanto indicato nella dichiarazione dei dati locali.

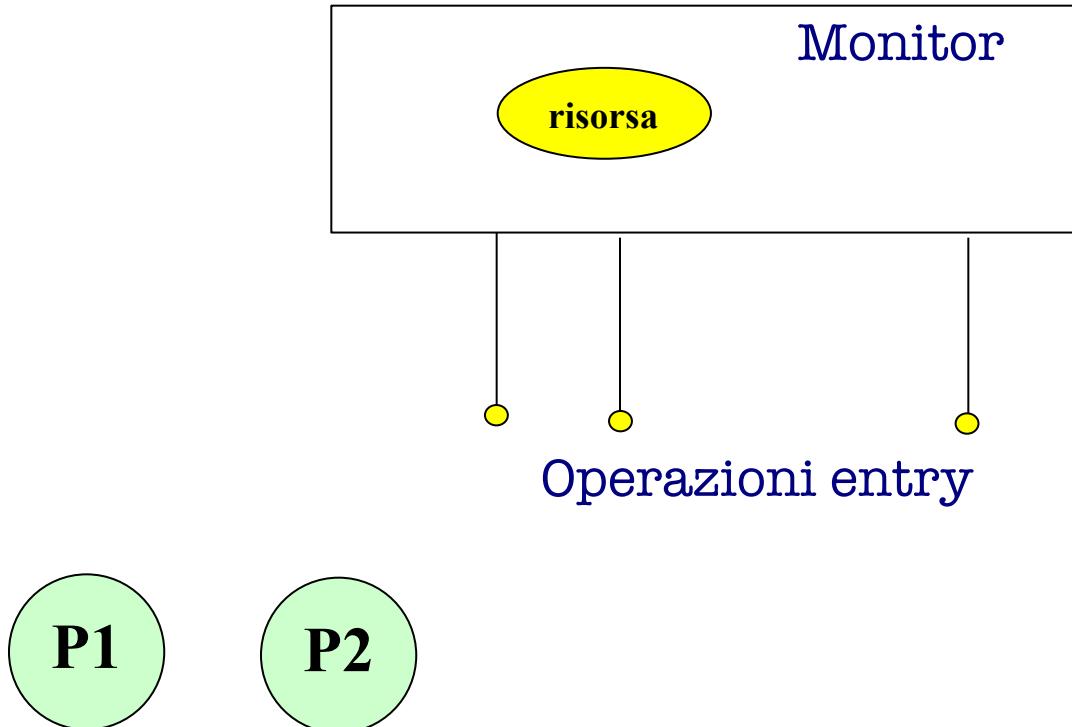
```
ris.opi(...);
```

- chiamata di una generica operazione dell'oggetto **ris**.

Uso del monitor

Solitamente, al **monitor** è associata una **risorsa**:

Scopo del monitor è **controllare l'accesso alla risorsa** da parte processi concorrenti, **in accordo a determinate politiche**. Le variabili locali definiscono lo stato della risorsa associata al monitor.



1 solo processo alla volta può “entrare” (i.e. eseguire una entry) nel monitor.

L'accesso alla risorsa avviene **tramite il monitor**, che garantisce **due livelli di sincronizzazione**:

1. Il **primo** garantisce che un solo processo alla volta possa aver accesso al monitor. Ciò è ottenuto mediante le operazioni **entry** che, per definizione, sono eseguite in **mutua esclusione** (eventuale sospensione dei processi nella coda **entry queue**).
2. Il **secondo** controlla **l'ordine** con il quale i processi hanno accesso alla risorsa. La procedura chiamata verifica il soddisfacimento di una condizione logica (**condizione di sincronizzazione**) che assicura l'ordinamento. Se la condizione logica non è soddisfatta, il processo viene posto **in attesa** ed il monitor viene liberato.

Monitor: sincronizzazione dei processi

Esempio: allocazione di una risorsa con priorità

```
monitor Risorsa()
{ boolean risorsa_libera=true;
  int turno=...;

...
entry void richiesta(int id)
{ if (turno!=id)
    <il processo esce dal monitor e aspetta>
    risorsa_libera=false;
    <attribuzione nuovo valore a turno>
}

entry void rilascio(int id)
{   risorsa_libera=true;
    <eventuale attribuzione nuovo valore a turno>
<risveglia il più prioritario tra i proc. in attesa>
}

}
```

Monitor: sincronizzazione dei processi

- Il **primo livello** di sincronizzazione (**mutua esclusione**) viene realizzato direttamente dal linguaggio: ogni primitiva **entry** è sempre mutuamente esclusiva.
- Il **secondo livello** di sincronizzazione viene **realizzato dal programmatore** in base alle politiche di accesso date, sfruttando un **nuovo strumento di sincronizzazione** associato al monitor: la **variabile condizione** (**condition**). In particolare:
 - L'accesso alla risorsa controllata dal monitor (da parte di un processo che esegue una entry) è vincolato al soddisfacimento di una **condizione di sincronizzazione**;
 - Nel caso in cui la condizione di sincronizzazione non sia verificata, il processo **si sospende liberando il monitor**; la sospensione del processo avviene tramite una **variabile condizione**.

Variabili condizione

Una variabile **condizione** rappresenta una **coda** nella quale i processi possono sospendersi (se la condizione di sincronizzazione non è verificata).

Definizione di una variabile **cond** di tipo condizione:

```
condition cond;
```

Operazioni sulle variabili condition:

- E' possibile applicare ad ogni variabile condizione **due operazioni**:

```
wait(cond);  
signal(cond);
```

Operazioni sulle variabili condizione

wait:

- L'esecuzione dell'operazione **wait(cond)** **sospende** il processo, introducendolo nella coda individuata dalla variabile **cond**, e il monitor viene liberato. Al risveglio, il processo riacquisisce l'accesso mutamente esclusivo al monitor e riprende l'esecuzione.

signal:

- L'esecuzione dell'operazione **signal(cond)** **riattiva** un processo in attesa nella coda individuata dalla variabile **cond**; se non vi sono processi in coda, non produce effetti.

Esempio: Monitor: uso di wait e signal

```
monitor Risorsa()
{   boolean risorsa_libera=true;
    condition C;
    int turno=...; // determina l'ordine di accesso
    ...
    entry void acquisizione(int id)
    {   while (turno!=id || risorsa_libera==false)
        C.wait();
        risorsa_libera=false;
    }

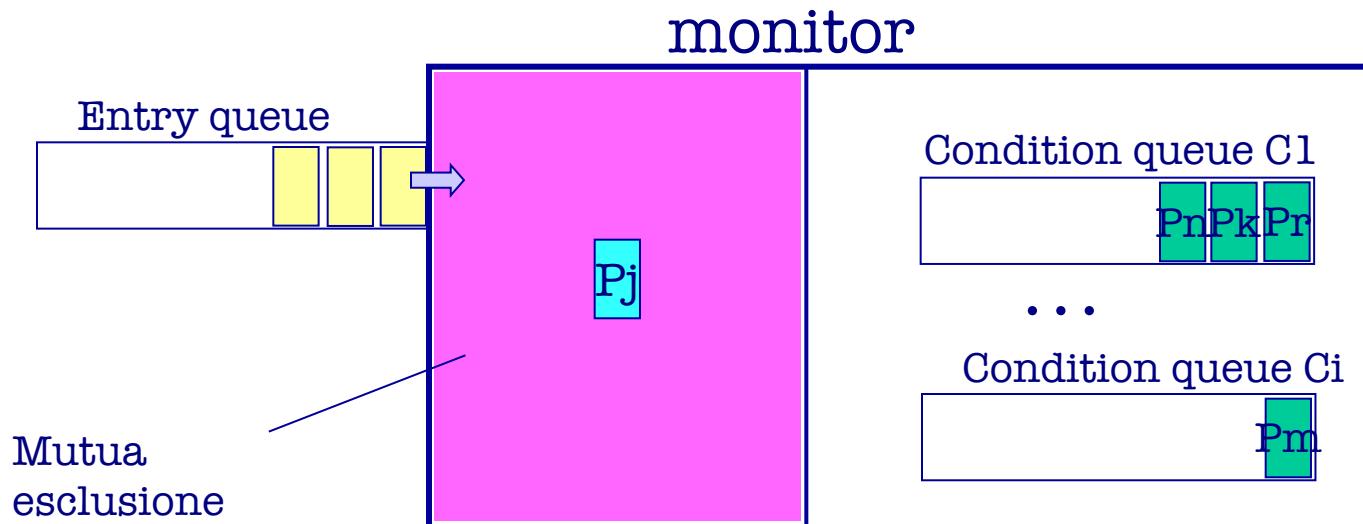
    entry void rilascio(int id)
    {   risorsa_libera=true;
        < attribuzione nuovo valore a turno>
        C.signal();
    }
}
```

Accesso al monitor: code

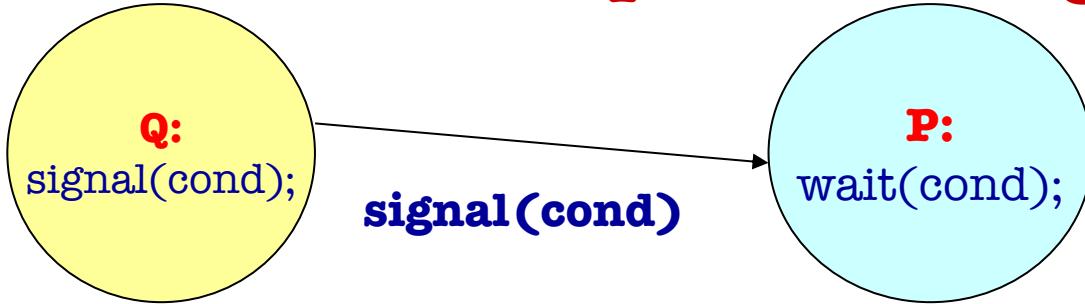
Il controllo nell'accesso al monitor viene esercitato tramite la sospensione dei processi in alcune code:

Primo livello (mutua esclusione): se un processo che vuole accedere al monitor (tramite un'operazione entry) lo trova occupato, esso viene sospeso nella **entry queue**

Secondo livello: se la condizione di sincronizzazione di un processo che esegue nel monitor (tramite un'operazione entry) non è soddisfatta, esso viene sospeso nella condition queue associata alla condizione di sincronizzazione (**condition queue**).



Semantiche dell'operazione signal



Come conseguenza della **signal**, entrambi i processi (quello «segnalante» **Q** e quello «segnalato» **P**), **possono proseguire la loro esecuzione**.

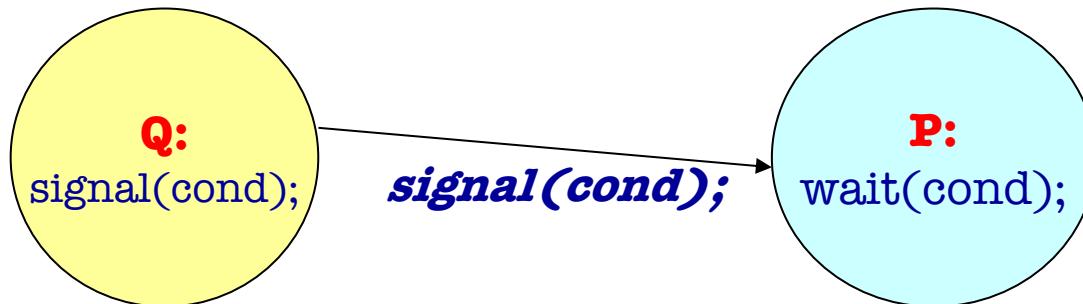
Il monitor, per definizione, limita a 1 il numero dei processi che possono eseguire al suo interno.

A chi dare la precedenza?

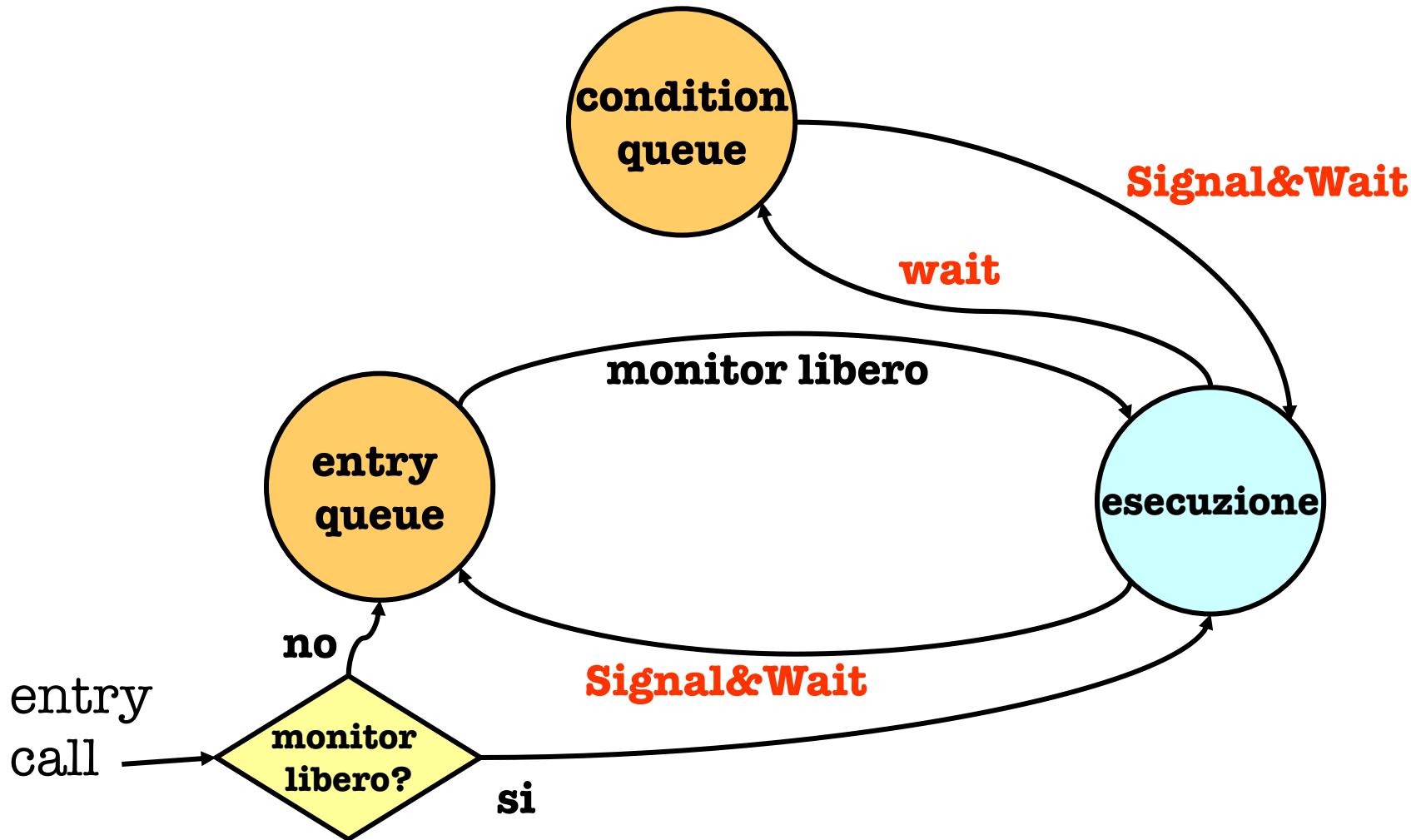
Semantiche dell'operazione signal

Possibili strategie (o semantiche):

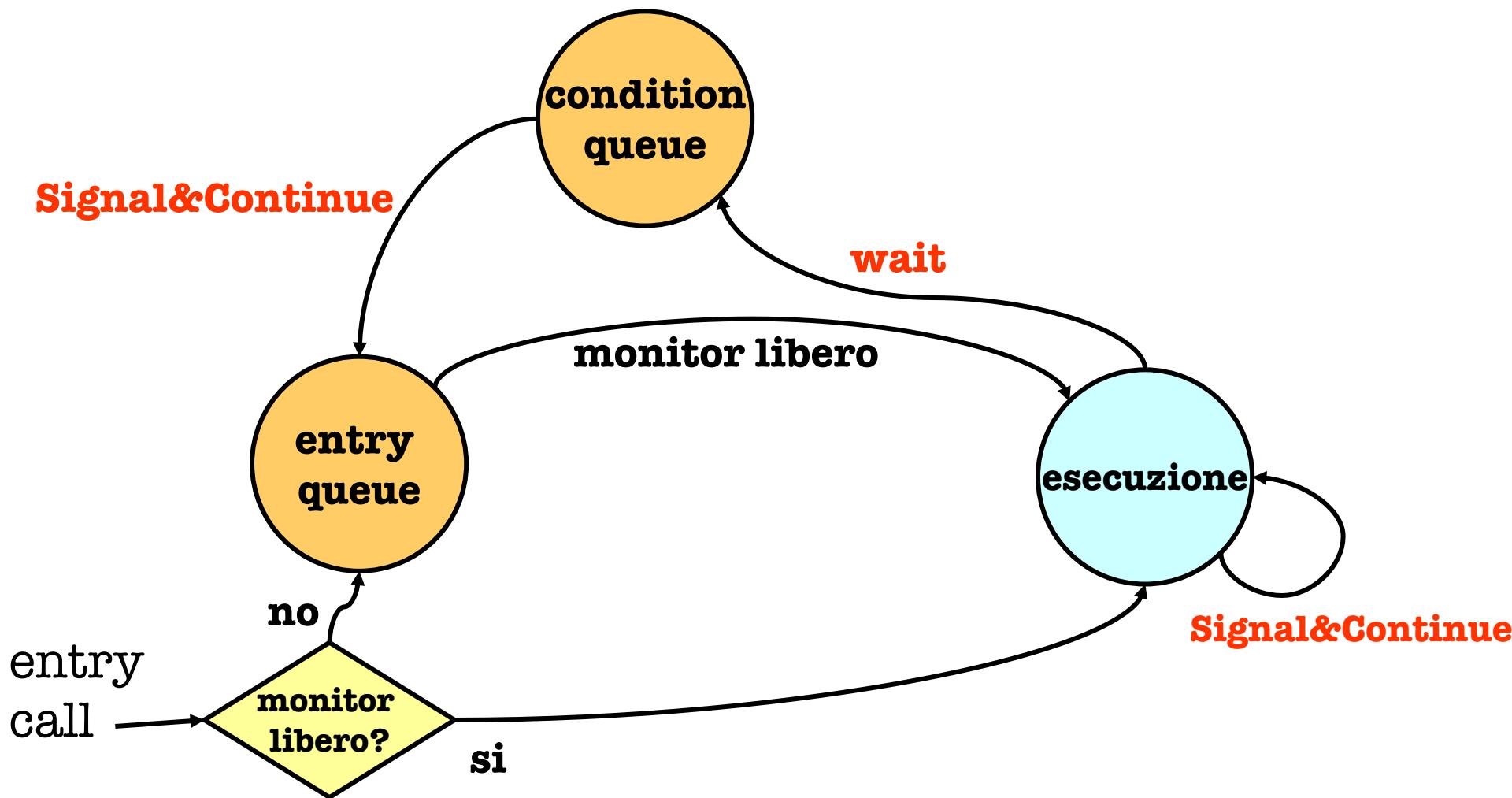
- **signal_and_wait:** **P riprende** immediatamente l'esecuzione ed il processo **Q viene sospeso nella entry queue.** (→ precedenza a P)
- **signal_and_continue:** **Q prosegue** la sua esecuzione mantenendo l'accesso esclusivo al monitor, dopo aver *riattivato* il processo **P**, il quale **si sospende nella entry queue**. (→ precedenza a Q)



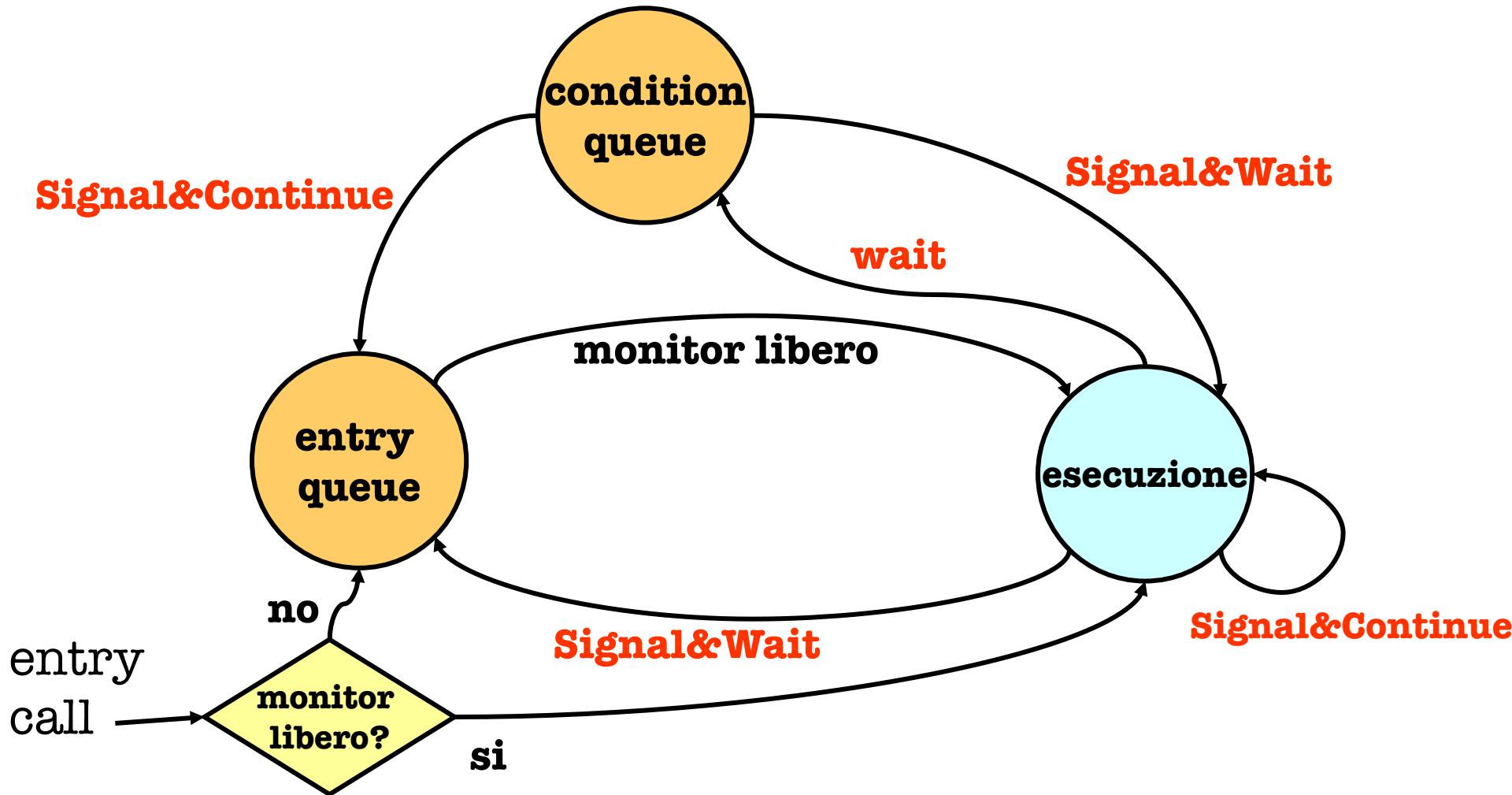
Semantiche della signal: signal&wait



Semantiche della signal: signal&continue



Semantiche della signal: diagramma complessivo



Signal_and_wait

- Q si sospende nella coda dei processi che attendono di usare il monitor (**entry queue**).
- Il primo processo ad operare nel monitor dopo la signal è certamente P:
 - ➡ Non è possibile che Q o altri processi possano modificare la condizione di sincronizzazione prima che P termini l'esecuzione della operazione entry.

signal_and_continue

- Il processo «segnalato» P viene trasferito dalla coda associata alla variabile condizione alla **entry_queue** e potrà rientrare nel monitor una volta che Q l'abbia rilasciato.
- Poiché altri processi possono entrare nel monitor prima di P, **questi potrebbero modificare la condizione di sincronizzazione** (lo stesso potrebbe fare Q).

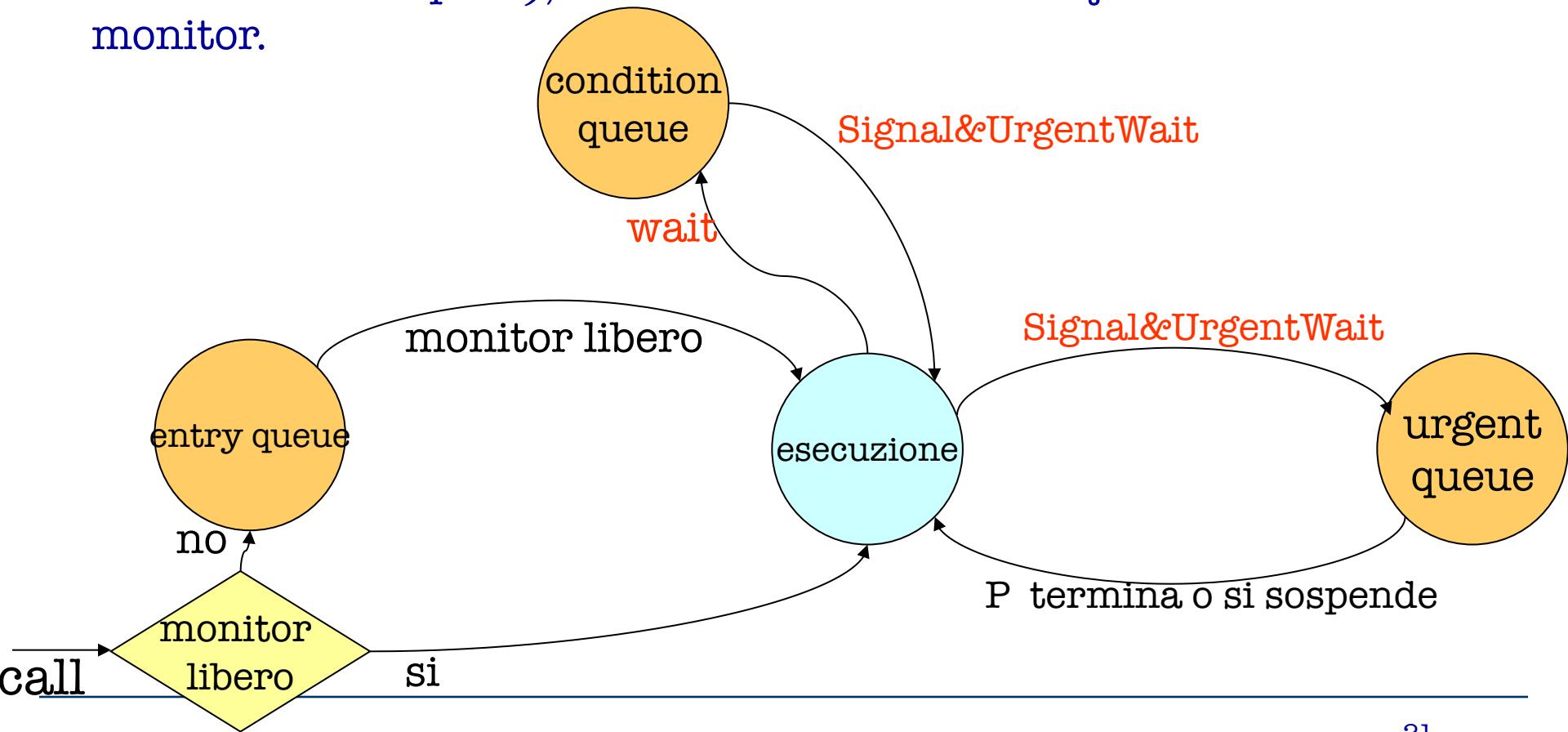
👉 E' pertanto **necessario** che, al rientro nel monitor, il processo segnalato P verifichi di nuovo la condizione di sincronizzazione:

```
while(!B) wait (cond);  
<accesso alla risorsa>
```

Varianti: signal_and_urgent_wait

signal_and_urgent_wait. E` una variante della signal_and_wait:

Q ha la priorità rispetto agli altri processi che aspettano di entrare nel monitor. Viene quindi sospeso in una coda interna al monitor (urgent queue). Quando P ha terminato la sua esecuzione (o si è nuovamente sospeso), trasferisce il controllo a Q senza liberare il monitor.



- Un caso particolare della **signal_and_urgent_wait** (e della signal_and_wait) si ha quando essa corrisponde ad una istruzione return:
signal_and_return.
- Il processo completa cioè la sua operazione con il risveglio del processo segnalato. Cede ad esso il controllo del monitor senza rilasciare la mutua esclusione.

- E' possibile anche **risvegliare tutti i processi** sospesi sulla variabile condizione utilizzando la :

signal_all

che è una variante della signal_and_continue.

- ➔ Tutti i processi risvegliati vengono messi nella entry_queue dalla quale, uno alla volta potranno rientrare nel monitor.

Esempio: monitor come gestore di risorse (mailbox)

Utilizziamo il monitor per risolvere il problema della comunicazione tra processi mediante un buffer di dimensione N (“**produttori e consumatori**”):

- ❑ la struttura dati che rappresenta il buffer **fa parte delle variabili locali al monitor** e quindi le operazioni *Send* e *Receive* possono accedere solo in modo **mutuamente esclusivo** a tale struttura.
- ❑ il monitor rappresenta il buffer dei messaggi (gestito in modo circolare)
- ❑ i processi Produttori (o Consumatori) inseriranno (o preleveranno) i messaggi mediante le funzioni entry *Send* (o *Receive*) definite nel monitor.

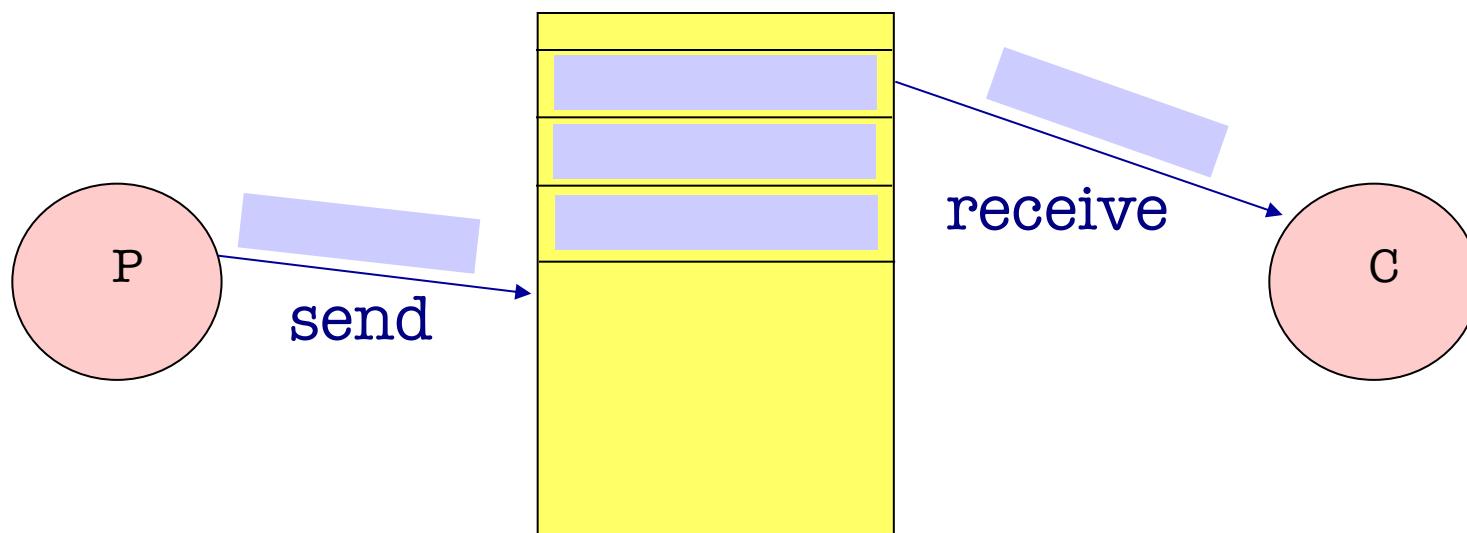
Semantiche della signal

La semantica della signal dipende dal linguaggio di programmazione utilizzato.

Ad esempio:

il linguaggio **Java** implementa la variabile condizione con semantica **signal&continue**

Esempio: Produttore Consumatore (buffer di capacita` n)



1. Il produttore non può inserire un messaggio nel buffer se questo è pieno.
2. Il consumatore non può prelevare un messaggio dal buffer se questo è vuoto

[***HP: semantica signal&wait***]

Soluzione con semantica Signal&Wait:

```
monitor buffer_circolare{
    messaggio buffer[N];
    int contatore=0; int testa=0; int coda=0;
    condition non_pieno;
    condition non_vuoto;

    /* procedure e funzioni entry: */
    entry void send(messaggio m){ /*proc. entry -> mutua esclusione*/
        if (contatore==N) non_pieno.wait;
        buffer[coda]=m;
        coda=(coda + 1)%N;
        ++contatore;
        non_vuoto.signal;
    }

    entry messaggio receive(){ /*proc. entry -> mutua esclusione*/
        messaggio m;
        if (contatore == 0) non_vuoto.wait;
        m=buffer[testa];
        testa=(testa + 1)%N;
        --contatore;
        non_pieno.signal;
        return m;}
}/* fine monitor */
```

Se la semantica fosse signal&continue ??

Soluzione con semantica Signal&Continue:

```
monitor buffer_circolare{
    messaggio buffer[N];
    int contatore=0; int testa=0; int coda=0;
    condition non_pieno;
    condition non_vuoto;

    /* procedure e funzioni entry: */
    entry void send(messaggio m) {
        while (contatore==N) //la condizione viene testata con while
            non_pieno.wait;
        buffer[coda]=m;
        coda=(coda + 1)%N;
        ++contatore;
        non_vuoto.signal;
    }

    entry messaggio receive() {
        messaggio m;
        while (contatore == 0) //la condizione viene testata con while
            non_vuoto.wait;
        m=buffer[testa];
        testa=(testa + 1)%N;
        --contatore;
        non_pieno.signal;
        return m;}
}/* fine monitor */
```

Esempio di uso del costrutto monitor

I filosofi a cena
(E. Dijkstra, 1965)

Il problema

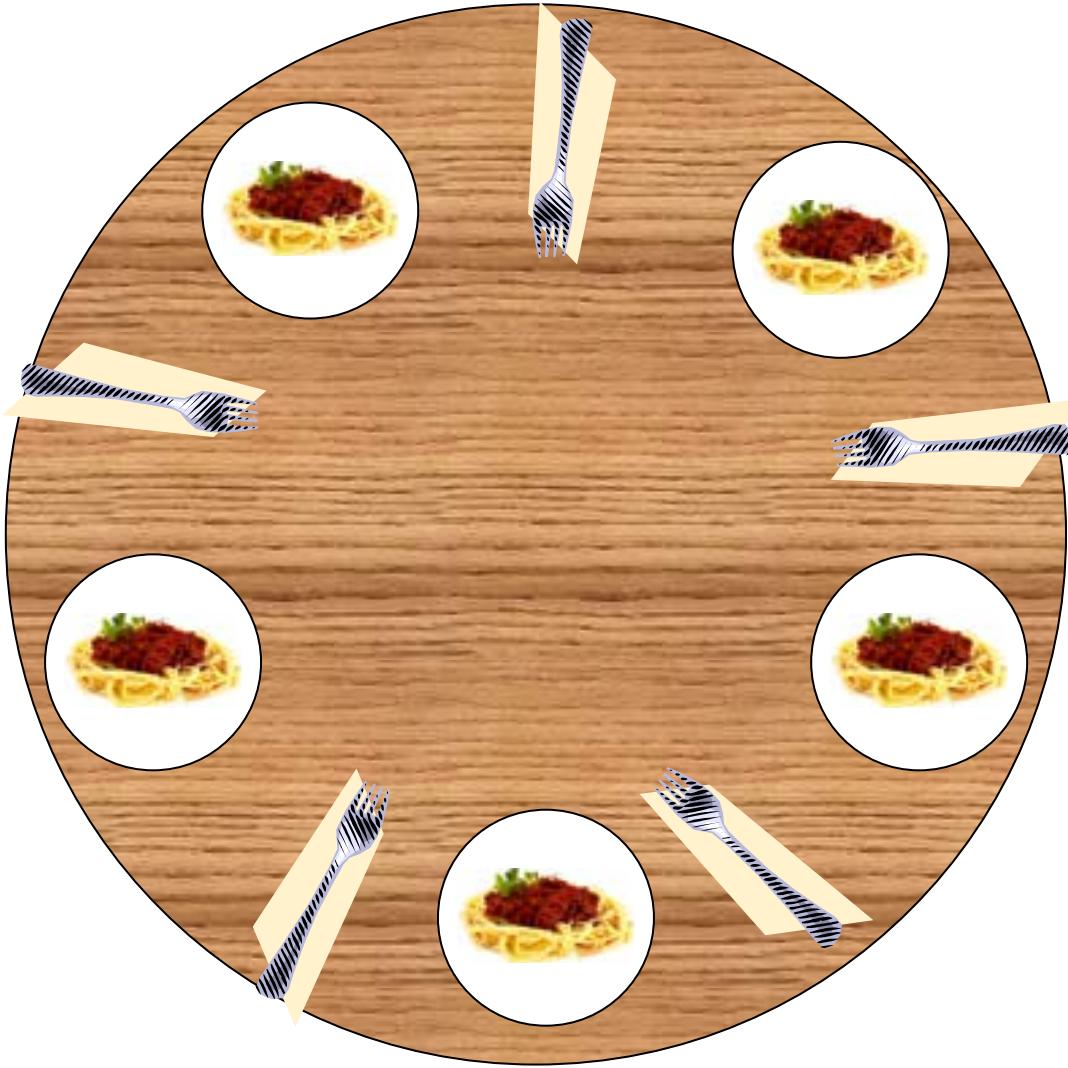
5 filosofi sono seduti attorno a un tavolo circolare; ogni filosofo ha un piatto di spaghetti tanto scivolosi che necessitano di 2 forchette per poter essere mangiati; sul tavolo vi sono in totale 5 forchette.

Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:

1. una fase in cui **pensa**,
2. una fase in cui **mangia**.

Rappresentando ogni filosofo con un thread, realizzare una politica di sincronizzazione che eviti situazioni di deadlock.

- [HP: semantica signal&continue]



Osservazioni

- i filosofi non possono mangiare tutti insieme: ci sono solo 5 forchette, mentre ne servirebbero 10;
- 2 filosofi vicini non possono mangiare contemporaneamente perche` condividono una forchetta e pertanto quando uno mangia, l'altro e` costretto ad attendere

Soluzione n. 1

Quando un filosofo ha fame:

1. prende la forchetta a sinistra del piatto
2. poi prende quella che a destra del suo piatto
3. mangia per un po'
4. poi mette sul tavolo le due forchette.

👉 **Possibilità di deadlock:** se tutti i filosofi afferrassero contemporaneamente la forchetta di sinistra, tutti rimarrebbero in attesa di un evento (il rilascio della forchetta alla propria destra) che non si potrà mai verificare .

Soluzione n.2

Ogni filosofo verifica se entrambe le forchette sono disponibili:

- in caso affermativo, acquisisce le due forchette (in modo atomico);
- in caso negativo, aspetta.

👉 in questo modo non si puo` verificare deadlock (non c'e` possesso e attesa)

Realizzazione soluzione 2

Quali processo?

- filosofo

Risorsa condivisa?

la tavola apparecchiata

-> definiamo la classe **tavola**, che rappresenta il monitor gestore delle forchette

Struttura Filosofo_i

tavola m; // istanza del monitor

```
process filosofo {
    while(true)
    {   <pensa...>
        m.prendiForchette(i);
        <mangia...>
        m.rilasciaForchette(i);
    }
}
```

Monitor

```
monitor tavola
{ int forchette[5] ={2,2,2,2,2};
//le forchette disponibili per ogni filosofo i
//inizialmente sono 2

condition codaF[5]; //1 coda per ogni filosofo i

// metodi entry :
entry void prendiForchette(int i){...}
entry void rilasciaForchette(int i){...}

// metodi privati:
int destra(int i){...}
int sinistra(int i) {...}
}
```

Metodi entry

```
entry void prendiForchette(int i)
{
    while (forchette[i] != 2)
        wait(codaF[i]);

    forchette[sinistra(i)]--;
    forchette[destra(i)]--;

}
```

```
entry void rilasciaForchette(int i)
{
    forchette[sinistra(i)]++;
    forchette[destra(i)]++;
    if (forchette[sinistra(i)]==2)
        signal(codaF[sinistra(i)]);
    if (forchette[destra(i)]==2)
        signal(codaF[destra(i)]);
}
```

Metodi privati

```
int destra(int i)
{ int ret;
  if (i==0)
    ret=NF-1;
  else ret=i-1;
  return ret;
}
```

```
int sinistra(int i)
{ int ret;
  ret=(i+1)%NF;
  return ret;
}
```

Realizzazione del costrutto monitor tramite semafori

HP: il kernel offre il semaforo come strumento “primitivo” di sincronizzazione.

Sia L un linguaggio concorrente che offre il costrutto monitor:

- 👉 Il compilatore del linguaggio L implementa il **costrutto linguistico monitor** mediante i semafori.

Realizzazione del costrutto monitor tramite semafori

Per ogni istanza di un monitor il compilatore del linguaggio concorrente prevede:

- un semaforo **mutex** inizializzato a 1 per la mutua esclusione delle operazioni entry del monitor:
 - la richiesta di un processo di eseguire un'operazione entry equivale all'esecuzione di una **p(mutex)**. Alla fine di ogni operazione entry viene eseguita una **v(mutex)**.
- per ogni variabile di tipo **condition**:
 - un semaforo **condsem** inizializzato a 0 sul quale il processo si può sospendere tramite una **p(condsem)**.
 - un contatore **condcount** inizializzato a 0 per tenere conto dei processi sospesi su condsem.

Implementazione con semantica Signal_and_continue:

Prologo di ogni operazione entry: **P(mutex)** ;

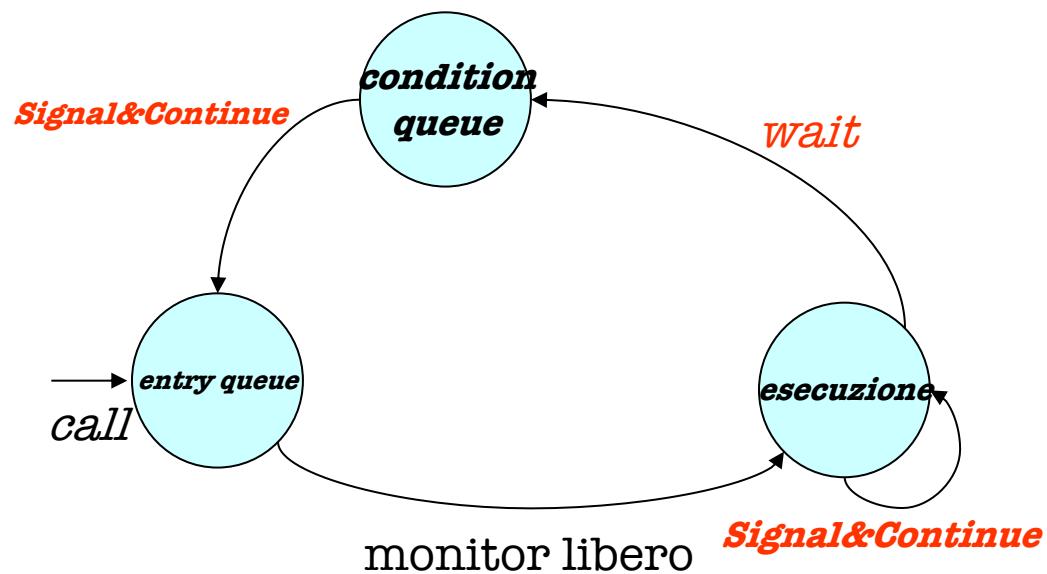
Epilogo di ogni operazione entry: **V(mutex)** ;

wait(cond)

```
{    condcount++;  
    V(mutex);  
    P(condsem);  
    P(mutex);  
}
```

signal(cond)

```
{    if (condcount>0)  
    {        condcount--;  
        V(condsem);  
    }  
}
```



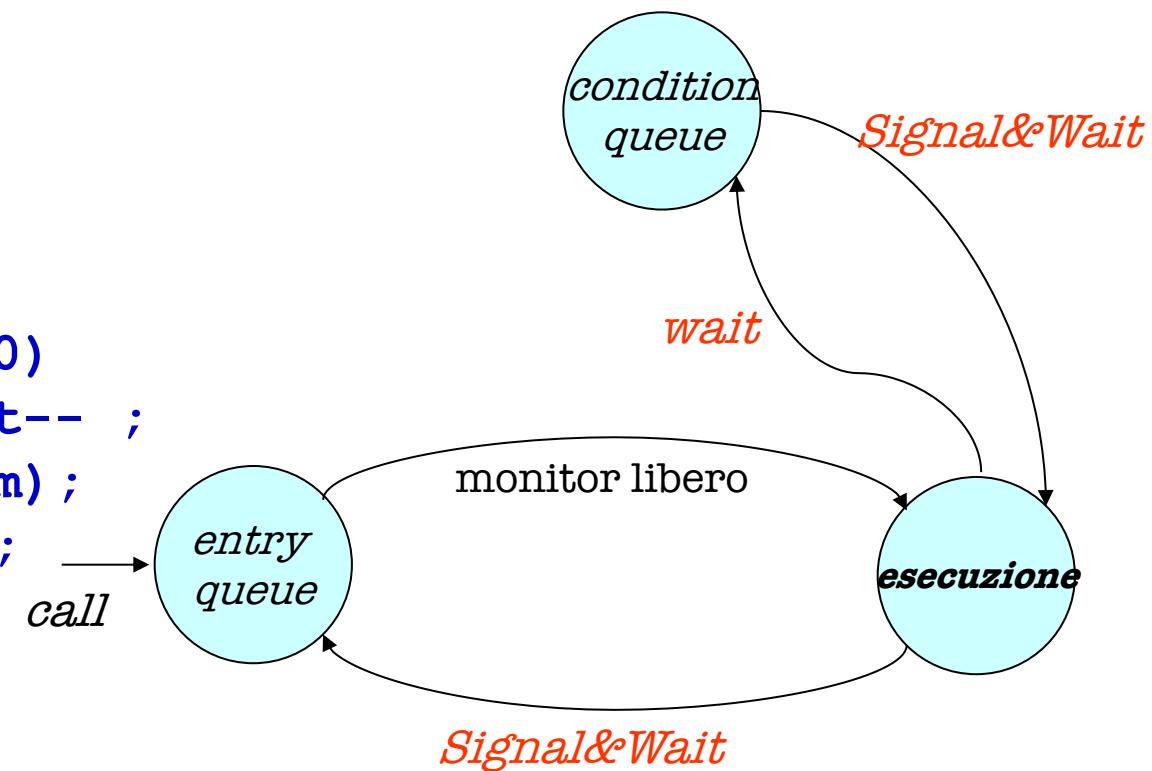
Signal_and_wait

Prologo di ogni operazione entry **P(mutex);**

Epilogo di ogni operazione entry **V(mutex);**

```
wait(cond)
{
    condcount++;
    V(mutex);
    P(condsem);
}
```

```
signal(cond)
{
    if (condcount>0)
    {
        condcount--;
        V(condsem);
        P(mutex);
    }
}
```



I Thread in Java

parte 2:

Sincronizzazione

Variabili condizione

- Nelle versioni più recenti di Java (**Java™ 2 Platform Standard Ed. 5.0**) esiste la possibilità utilizzare le **variabili condizione**. Ciò è ottenibile tramite l'uso un'apposita interfaccia (definita in **java.util.concurrent.locks**) :

```
public interface Condition{  
    //Public instance methods  
    void await () throws InterruptedException;  
    void signal();  
    void signalAll();  
}
```

- dove i metodi **await**, **signal**, e **signalAll** sono del tutto equivalenti ai metodi wait, signal e signalAll riferiti in genere alle variabili condizione
- le condition Java sono implementate con **semantica “signal_and_continue”**

Mutua esclusione: lock

- Oltre a metodi/blocchi synchronized, la versione 5.0 di Java prevede la possibilita` di utilizzare esplicitamente il concetto di *lock*, mediante l'interfaccia (definita in `java.util.concurrent.locks`) :

```
public interface Lock{  
    //Public instance methods  
    void lock();  
    void unlock();  
    Condition newCondition();  
}
```

Uso di Variabili Condizione

Ad ogni variabile condizione deve essere associato un lock, che:

- al momento della sospensione del thread mediante `await` il lock verrà liberato;
- al risveglio di un thread, il lock verrà automaticamente rioccupato.

→ La creazione di una condition deve essere effettuata mediante il metodo `newCondition` del lock associato ad essa.

In pratica, **per creare un oggetto Condition :**

```
Lock lockvar=new Reentrantlock(); //Reentrantlock è una
                                  // classe che implementa
                                  // l'interfaccia Lock
Condition C=lockvar.newCondition(); //C è una condition
                                    // associata al lock lockvar
```

Monitor in Java

Java non offre un costrutto equivalente al monitor, ma con gli strumenti visti possiamo definire **classi** che implementano il concetto di monitor:

Dati:

- ❑ le **variabili condizione**
- ❑ **1 lock per la mutua esclusione** dei metodi "entry", da associare a tutte le variabili condizione
- ❑ variabili interne: stato delle risorse gestite

Metodi:

- ❑ metodi **public** ("entry")
- ❑ metodi privati
- ❑ costruttore

Esempio: gestione di buffer circolare

```
public class Mailbox // monitor
{ //Dati:
private int[] contenuto;
private int contatore,testa,coda;
private Lock lock= new ReentrantLock();
private Condition non_pieno= lock.newCondition();
private Condition non_vuoto= lock.newCondition();

//Costruttore:
public Mailbox( ) {
contenuto=new int[N];
contatore=0;
testa=0;
coda=0;
}
```

```
//metodi "entry":  
  
public int preleva() throws InterruptedException  
{ int elemento;  
    lock.lock();  
    try  
    { while (contatore== 0)  
        non_vuoto.await();  
        elemento=contenuto[testa];  
        testa=(testa+1)%N;  
        --contatore;  
        non_pieno.signal ( );  
    } finally{lock.unlock();}  
    return elemento;  
}
```

```
public void deposita (int valore) throws
    InterruptedException
{ lock.lock();
try
{   while (contatore==N)
        non_pieno.await();
    contenuto[coda] = valore;
    coda=(coda+1)%N;
    ++contatore;
    non_vuoto.signal( );
} finally{ lock.unlock(); }
}
```

Programma di test:

```
public class Produttore extends Thread
{ int messaggio;
Mailbox m;
public Produttore(Mailbox M) {this.m =M; }
public void run()
{ while(1)
{ <produci messaggio>
  m.deposita(messaggio);
}
}
}

public class Consumatore extends Thread
{ int messaggio;
Mailbox m;
public Consumatore(Mailbox M) {this.m =M; }
public void run()
{ while(1)
{ messaggio=m.preleva();
  <consuma messaggio>
}
}
}
```

```
public class BufferTest{  
  
    public static void main(String args[])  
    {   Mailbox M=new Mailbox();  
        Consumatore C=new Consumatore(M);  
        Produttore P=new Produttore(M);  
        C.start();  
        P.start();  
        ...  
    }  
}
```

I filosofi a cena

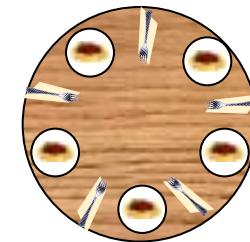
(E. Dijkstra, 1965)

Il problema

5 filosofi sono seduti attorno a un tavolo circolare; ogni filosofo ha un piatto di spaghetti tanto scivolosi che necessitano di 2 forchette per poter essere mangiati; sul tavolo vi sono in totale 5 forchette.

Ogni filosofo ha un comportamento ripetitivo, che alterna due fasi:

- una fase in cui **pensa**,
- una fase in cui **mangia**.



Rappresentando ogni filosofo con un thread, realizzare una politica di sincronizzazione che eviti situazioni di deadlock.

Soluzione corretta

Ogni filosofo verifica se entrambe le forchette sono disponibili:

- in caso affermativo, acquisisce le due forchette (in modo atomico);
- in caso negativo, aspetta.

[in questo modo **si previene il deadlock**: non c'e` possesso e attesa]

Realizzazione

Quali thread?

- I filosofi: definiamo la **classe filosofo**

Risorsa condivisa?

la tavola apparecchiata

-> definiamo la **classe tavola**, che rappresenta il monitor allocatore delle forchette

Struttura Filosofo_i

```
public class filosofo extends Thread
{ tavola m;
  int i;
  public filosofo(tavola M, int id){this.m =M;this.i=id;}
  public void run()
  { try{
    while(true)
    { System.out.print("Filosofo "+ i+" pensa....\n");
      m.prendiForchette(i);
      System.out.print("Filosofo "+ i+" mangia....\n");
      sleep(8);
      m.rilasciaForchette(i);
      sleep(100);
    }
  }catch(InterruptedException e) {}
}
```

Monitor: tavola

```
public class tavola
{ //Costanti:
    private final int NF=5;      // num forchette/filosofi
    //Dati:
    private int []forchette=new int[NF]; // forch. disp. per
                                         // ogni filosofo i
    private Lock lock= new ReentrantLock();
    private Condition []codaF= new Condition[NF];//code fil.

    //Costruttore:
    public tavola( ) {
        int i;
        for(i=0; i<NF; i++)
            codaF[i]=lock.newCondition();
        for(i=0; i<NF; i++)
            forchette[i]=2;
    }
    // metodi public e metodi privati:...}
```

Metodi public

```
public void prendiForchette(int i) throws  
    InterruptedException  
{ lock.lock();  
    try  
{  
        while (forchette[i]!=2)  
            codaF[i].await();  
  
        forchette[sinistra(i)]--;  
        forchette[destra(i)]--;  
  
    } finally{ lock.unlock();}  
    return;  
}
```

```
public void rilasciaForchette(int i) throws  
InterruptedException  
{ lock.lock();  
try  
{  
    forchette[sinistra(i)]++;  
    forchette[destra(i)]++;  
    if (forchette[sinistra(i)]==2)  
        codaF[sinistra(i)].signal();  
    if (forchette[destra(i)]==2)  
        codaF[destra(i)].signal();  
  
} finally{ lock.unlock();}  
return;  
}
```

Metodi privati

```
int destra(int i)
{ int ret;
  ret=(i==0? NF-1: (i-1));
  return ret;
}
```

```
int sinistra(int i)
{ int ret;
  ret=(i+1)%NF;
  return ret;
}
```

Programma di test

```
import java.util.concurrent.*;  
  
public class Filosofi {  
    public static void main(String[] args) {  
        int i;  
        tavola M=new tavola();  
        filosofo []F=new filosofo[5];  
        for(i=0;i<5;i++)  
            F[i]=new filosofo(M, i);  
        for(i=0;i<5;i++)  
            F[i].start();  
    }  
}
```

GESTIONE DELLE PERIFERICHE D' INGRESSO/USCITA

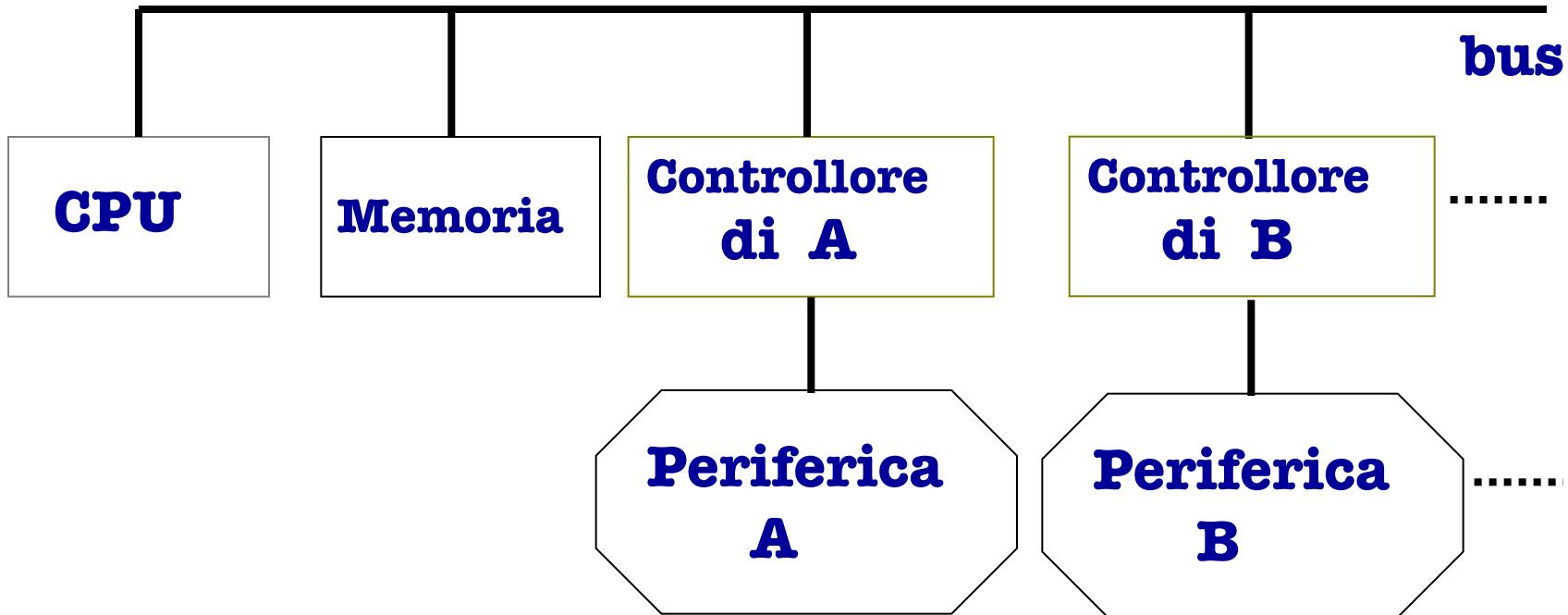
- **Compiti del sottosistema di I/O**
- **Architettura del sottosistema di I/O**
- **Gestore di un dispositivo di I/O**

COMPITI DEL SOTTOSISTEMA DI I/O

- 1. Nascondere al programmatore i dettagli delle interfacce hardware dei dispositivi;**
- 2. Omogeneizzare la gestione di dispositivi diversi;**
- 3. Gestire i malfunzionamenti che si possono verificare durante un trasferimento di dati;**
- 4. Definire lo spazio dei nomi (*naming*) con cui vengono identificati i dispositivi;**
- 5. Garantire la corretta sincronizzazione tra ogni processo applicativo che ha attivato un trasferimento dati e l'attività del dispositivo.**

COMPITI DEL SOTTOSISTEMA DI I/O

1. Nascondere al programmatore i dettagli delle interfacce hardware dei dispositivi



COMPITI DEL SOTTOSISTEMA DI I/O

2) Omogeneizzare la gestione di dispositivi diversi

TIPOLOGIE DI DISPOSITIVI

- Dispositivi a carattere (es. tastiera, stampante, mouse,...)
- Dispositivi a blocchi (es. dischi, nastri, ..)
- Dispositivi speciali (es. timer)

COMPITI DEL SOTTOSISTEMA DI I/O

2) Omogeneizzare la gestione di dispositivi diversi

dispositivo	velocità di trasferimento
tastiera	10 bytes/sec
mouse	100 bytes/sec
modem	10 Kbytes/sec
linea ISDN	16 Kbytes/sec
stampante laser	100 Kbytes/sec
scanner	400 Kbytes/sec
porta USB1	1.5 Mbytes/sec
disco IDE	5 Mbytes/sec
CD-ROM	6 Mbytes/sec
Fast Etherneet	12.5 Mbytes/sec
FireWire (IEEE 1394)	50 Mbytes/sec
monitor XGA	60 Mbytes/sec
Ethernet gigabit	125 Mbytes/sec

COMPITI DEL SOTTOSISTEMA DI I/O

3. Gestire i malfunzionamenti che si possono verificare durante un trasferimento di dati

TIPOLOGIE DI GUASTI

- Guasti **transitori** (es. disturbi elettromagnetici durante un trasferimento dati);
- Guasti **permanenti** (es. rottura di una testina di lettura/scrittura di un disco).
- **Eventi** eccezionali (es. mancanza di carta sulla stampante, end-of-file);

COMPITI DEL SOTTOSISTEMA DI I/O

4. Definire lo spazio dei nomi (*naming*) con cui vengono identificati i dispositivi

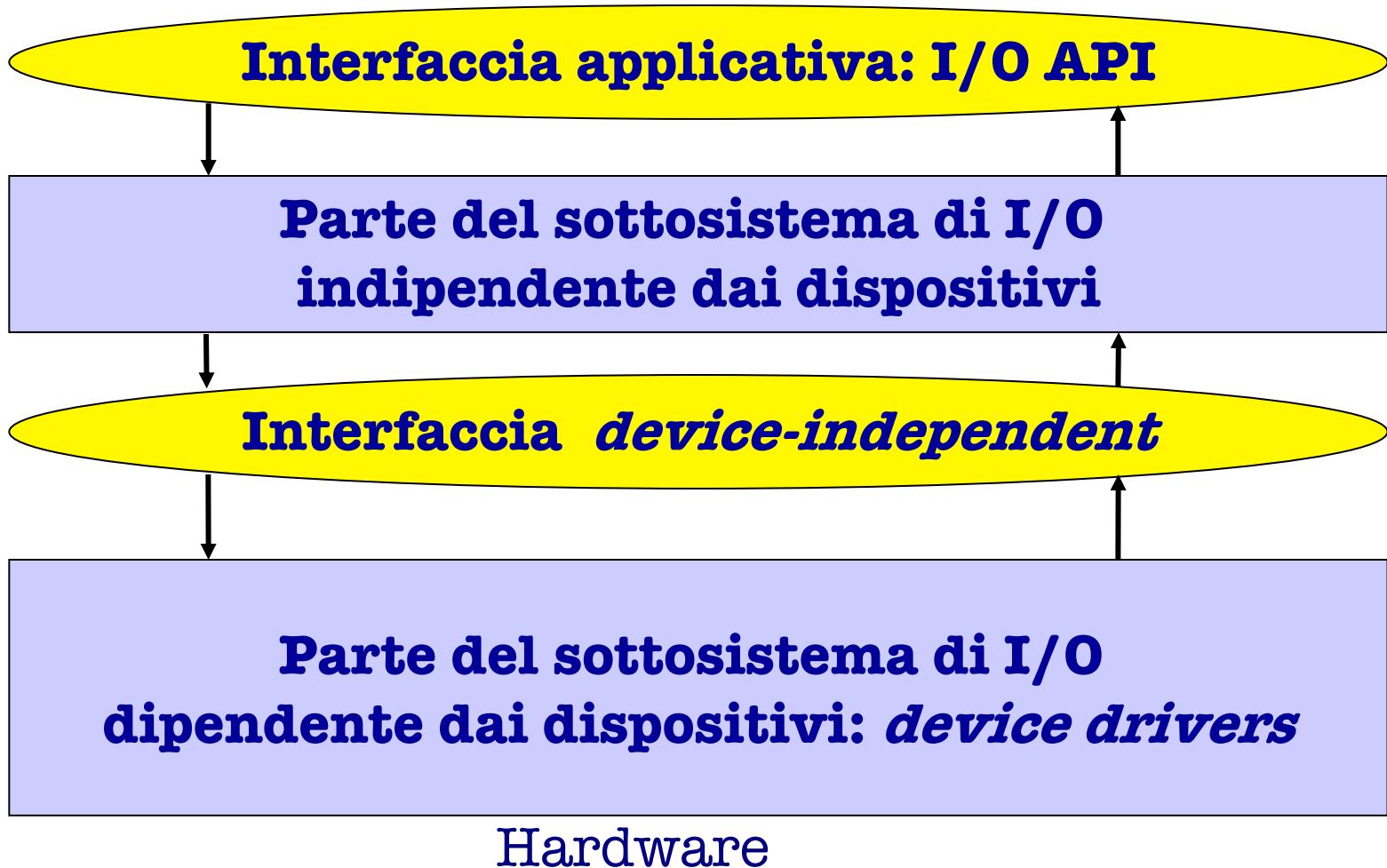
- Uso di **nomi simbolici** da parte dell' **utente** (*I/O API Input/Output Application Programming Interface*);
- Uso di **identificatori unici** (es: valori numerici interi) all' interno del sistema per identificare in modo univoco i dispositivi;
- Spesso: uniformità col meccanismo di *naming* del file-system.

COMPITI DEL SOTTOSISTEMA DI I/O

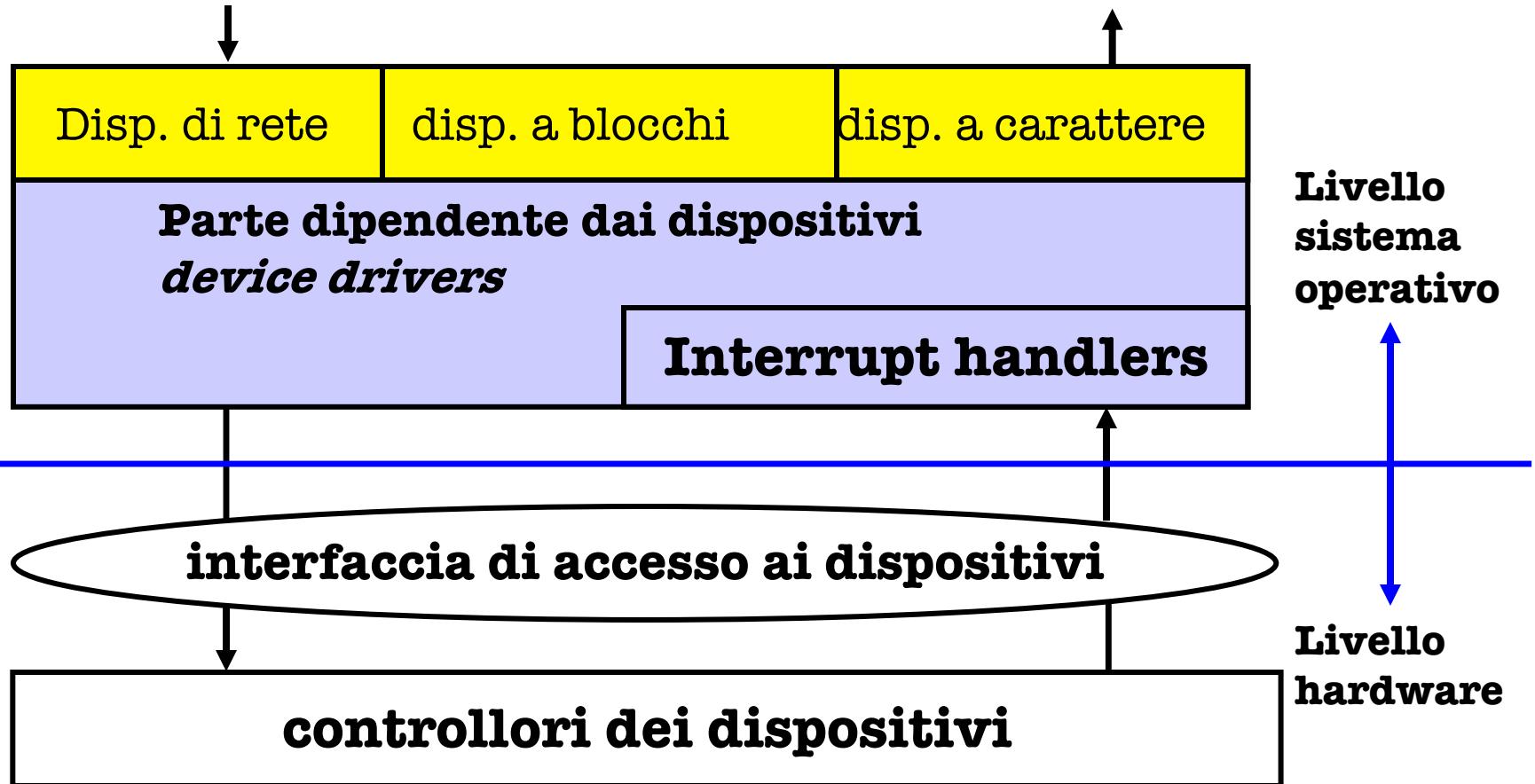
5. Garantire la corretta sincronizzazione tra un processo applicativo che ha attivato un trasferimento dati e l' attività del dispositivo.

- Gestione **sincrona** dei trasferimenti: un processo applicativo attiva un dispositivo e si blocca fino al termine del trasferimento;
- Gestione **asincrona** dei trasferimenti: un processo applicativo attiva un dispositivo e prosegue senza bloccarsi;
- Necessità di gestire la “*bufferizzazione*” dei dati.

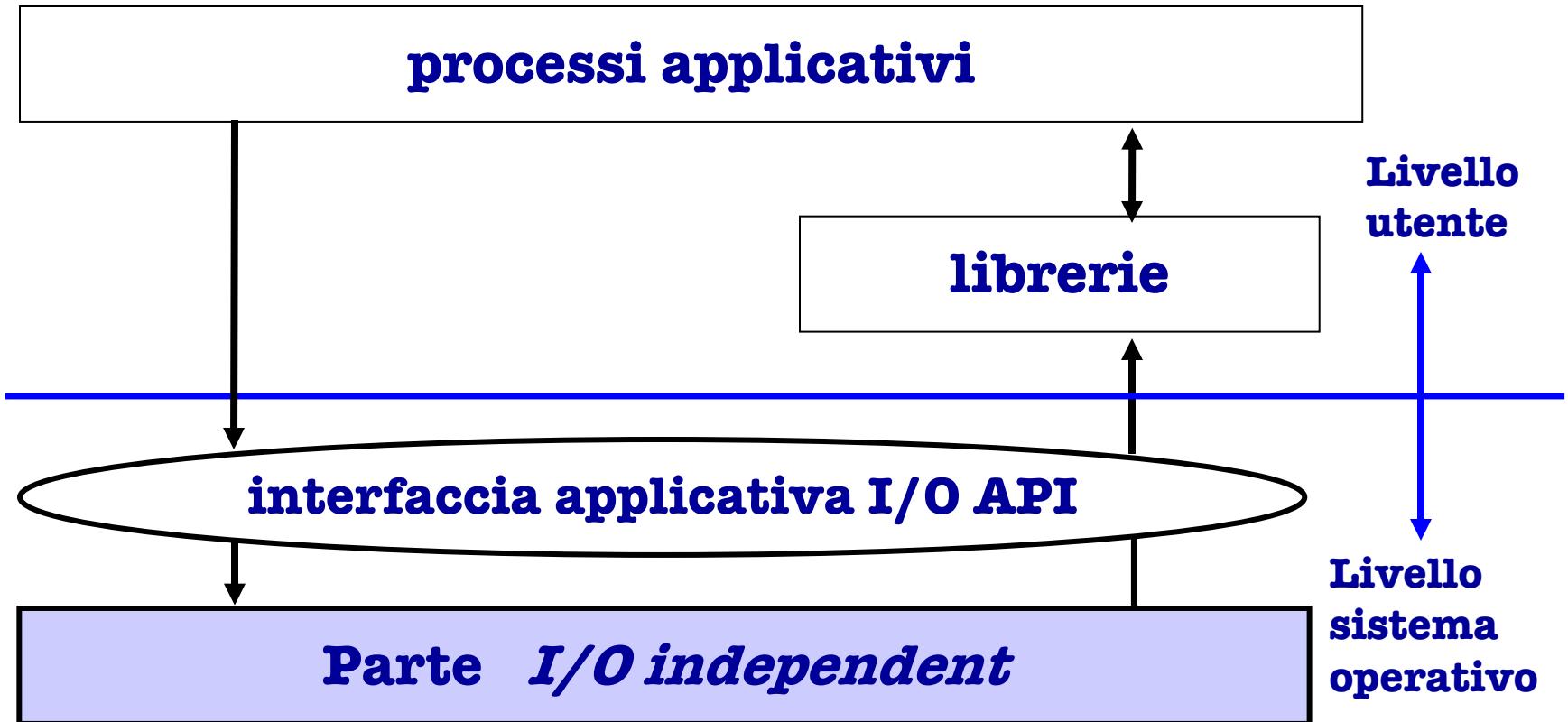
ARCHITETTURA DEL SOTTOSISTEMA DI I/O



ARCHITETTURA DEL SOTTOSISTEMA DI I/O: parte dipendente dai dispositivi



ARCHITETTURA DEL SOTTOSISTEMA DI I/O: parte indipendente dai dispositivi



LIVELLO INDEPENDENTE DAI DISPOSITIVI

FUNZIONI

- **Naming**
- **Buffering**
- **Gestione malfunzionamenti**
- **Allocazione dei dispositivi ai processi applicativi**

BUFFERING

Per ogni operazione di I/O il sistema operativo riserva un'area di memoria "tampone" (buffer), per contenere i dati oggetto del trasferimento.

Motivazioni:

- **differenza di velocita` tra processo e periferica: disaccoppiamento**
- **quantita` di dati da trasferire (es. dispositivi a blocchi): il processo puo` richiedere il trasferimento di una quantita` di informazioni inferiore a quella del blocco**

BUFFERING

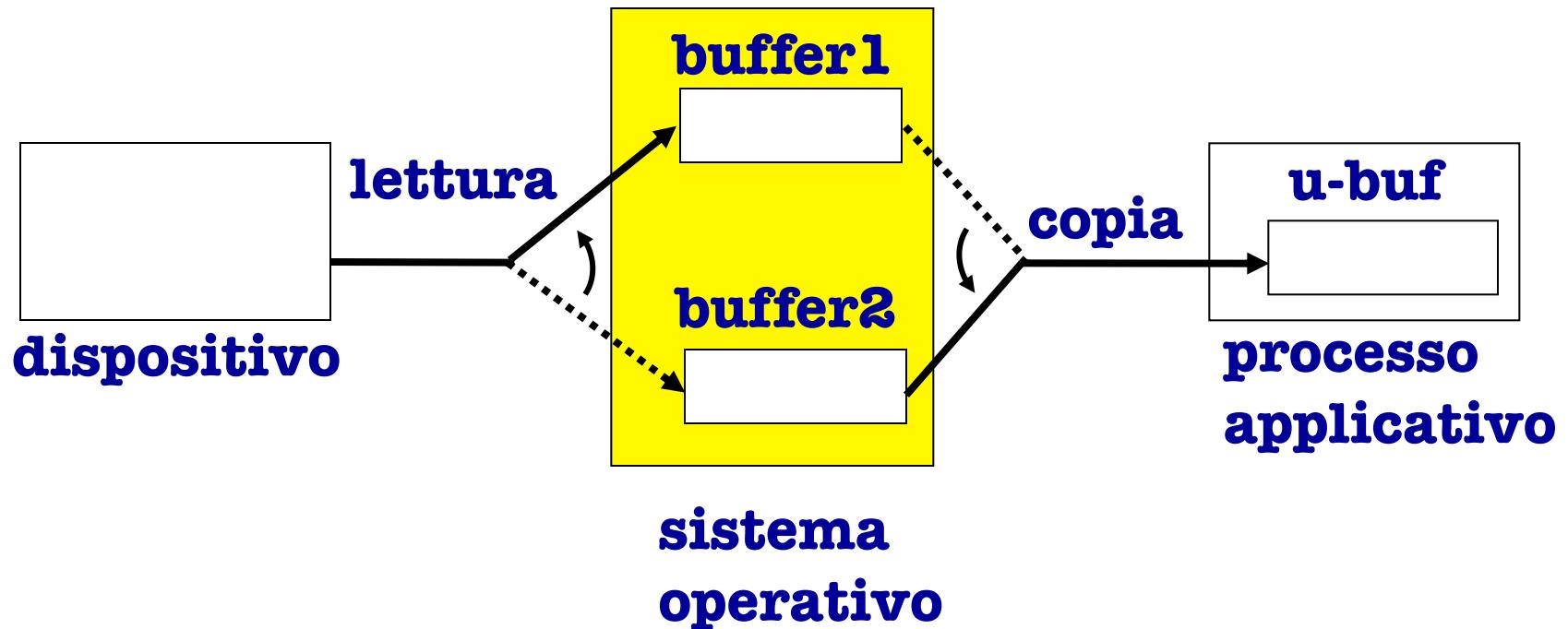
ES. operazione di lettura con singolo buffer



- **Buffer: area tampone nella memoria del sistema operativo**
- **u-buf: area tampone (variabile) nello spazio di indirizzamento del processo applicativo**

BUFFERING

ES. operazione di lettura con doppio buffer



GESTIONE MALFUNZIONAMENTI

- **Tipi di eventi anomali:**
 - Eventi generati a questo livello (es. tentativo di accesso a un dispositivo inesistente).
 - Eventi propagati dal livello inferiore (es. guasto HW temporaneo o permanente);
- **Gestione degli eventi anomali:**
 - Risoluzione del problema (mascheramento dell' evento anomalo);
 - Gestione parziale e propagazione a livello applicativo.

ALLOCAZIONE DEI DISPOSITIVI

- **Dispositivi condivisi da utilizzare in mutua esclusione;**
- **Dispositivi dedicati ad un solo processo (server) a cui i processi client possono inviare messaggi di richiesta di servizio;**
- **Tecniche di spooling (dispositivi virtuali).**

LIVELLO DIPENDENTE DAI DISPOSITIVI

Funzioni:

- fornire i gestori dei dispositivi (*device drivers*)
- offrire al livello superiore l' insieme delle funzioni di accesso ai dispositivi (tramite un'interfaccia “*device-independent*”).

Ad es. , dispositivo di INPUT:

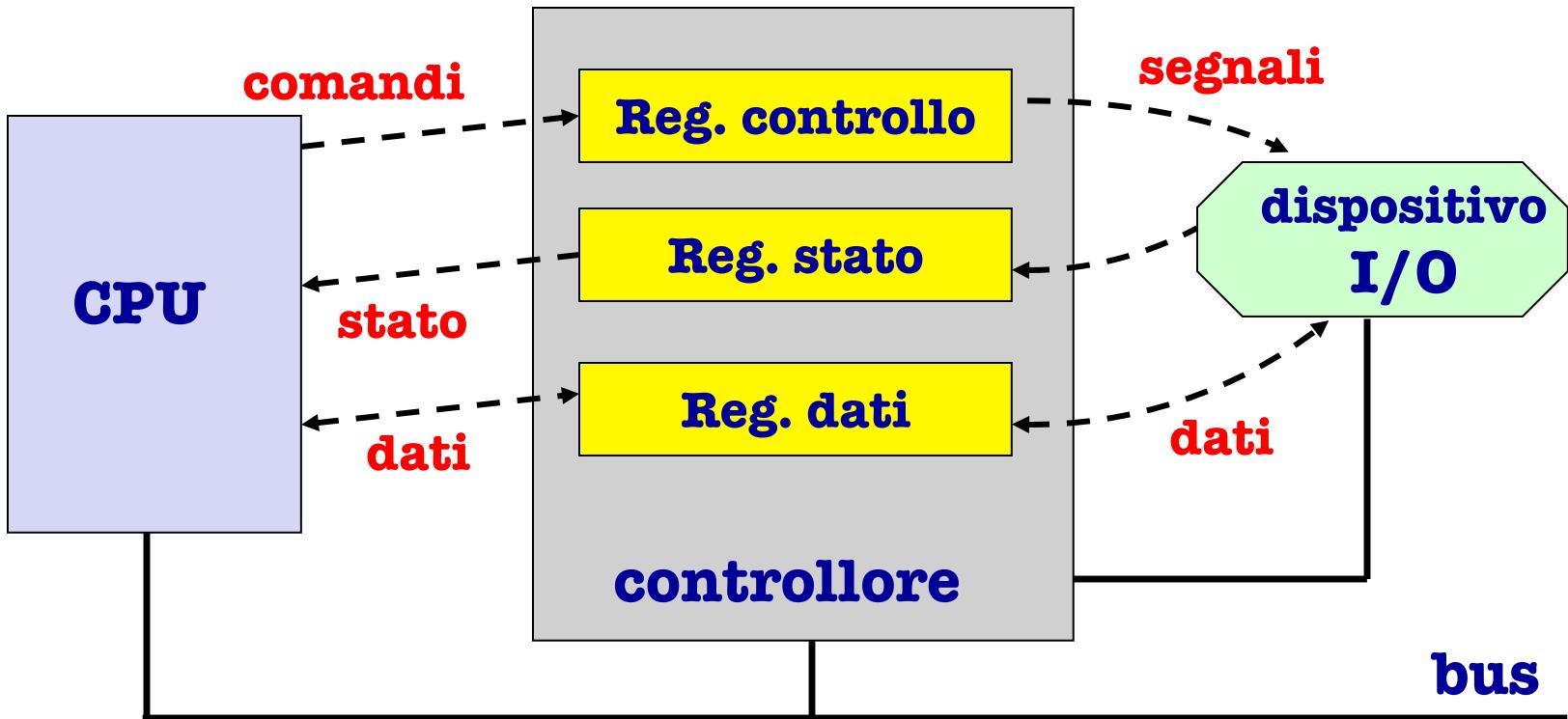
N=_read (disp, buffer, nbytes)

nome unico
del dispositivo

Buffer di sistema

Interazione DISPOSITIVO-CPU

Schema semplificato di un controllore (I/O)



Controllore di un **DISPOSITIVO** di I/O

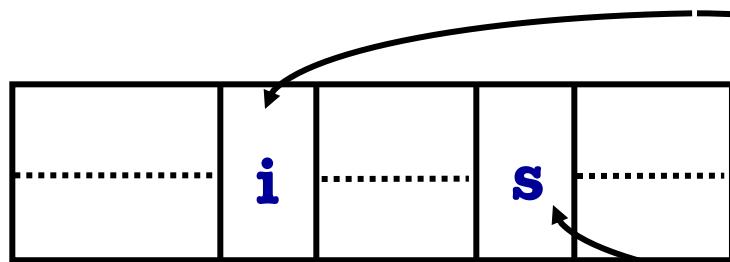
Registro di controllo: viene scritto dalla CPU per controllare la periferica; ogni comando da impartire alla periferica viene scritto, tramite il driver, nel registro di controllo. La periferica viene attivata tramite il settaggio del bit di **start**.

Registro di stato: la periferica usa il registro di stato per comunicare l'esito di ogni comando eseguito ed, eventualmente, per notificare eventuali errori occorsi. Il termine del comando viene notificato tramite il bit di flag.

Registro Dati: viene utilizzato per il trasferimento dei dati letti/scritti dal dispositivo

Controllore di UN DISPOSITIVO

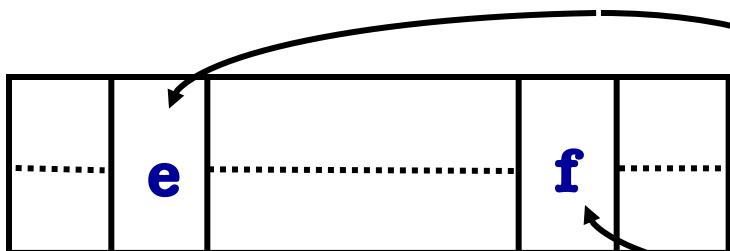
Registri di stato e controllo



Registri di controllo

i: bit di abilitazione alle interruzioni

s: bit di start



Registri di stato

e: bit di condizioni di errore

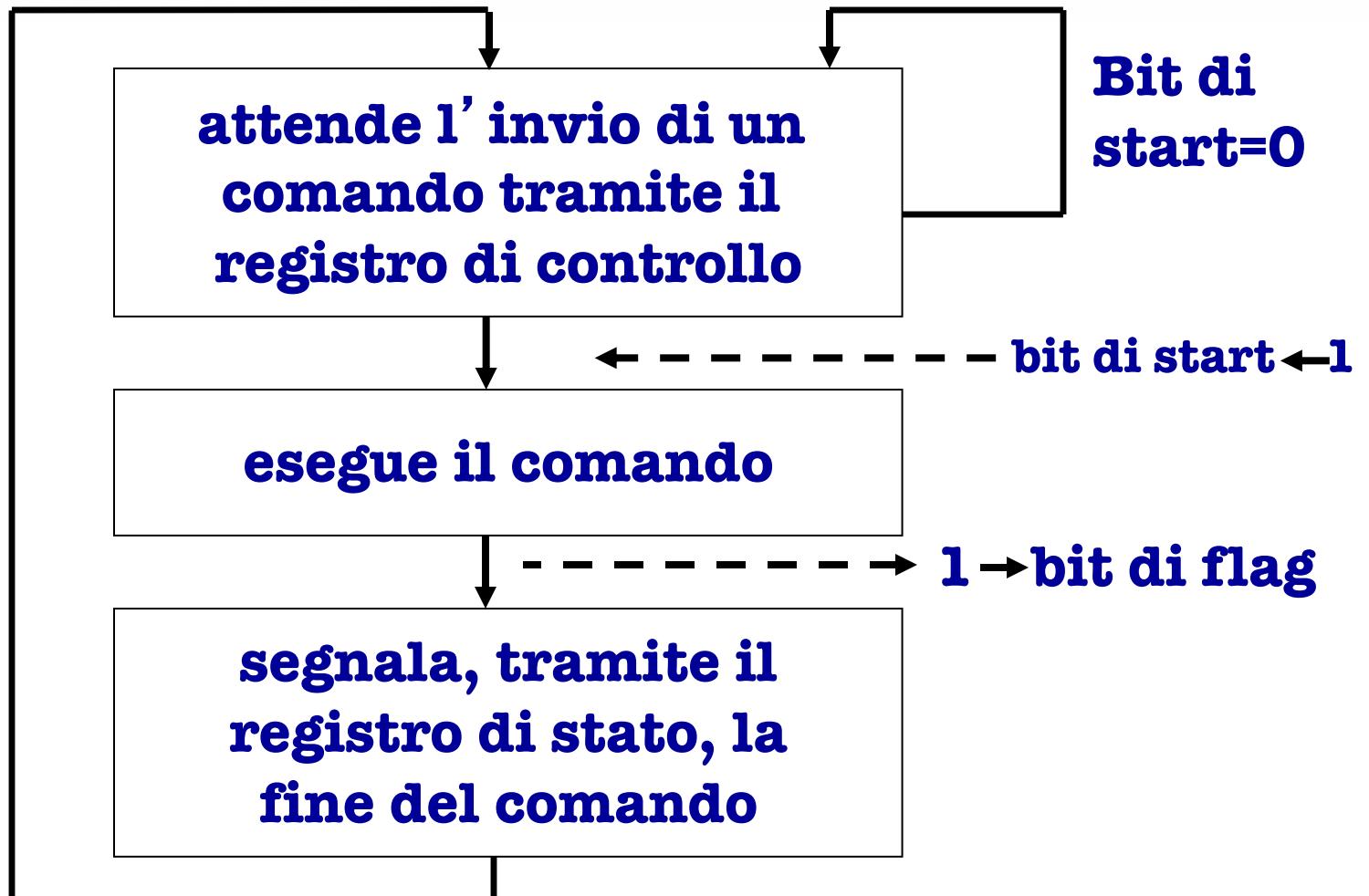
s: bit di flag

Modalità di gestione di un dispositivo

- Nella gestione di ogni dispositivo, la sincronizzazione tra CPU e periferica può avvenire secondo 2 modelli alternativi:
 1. Gestione a **controllo di programma** (o polling)
 2. Gestione basata su **interruzioni**

Gestione a controllo di programma

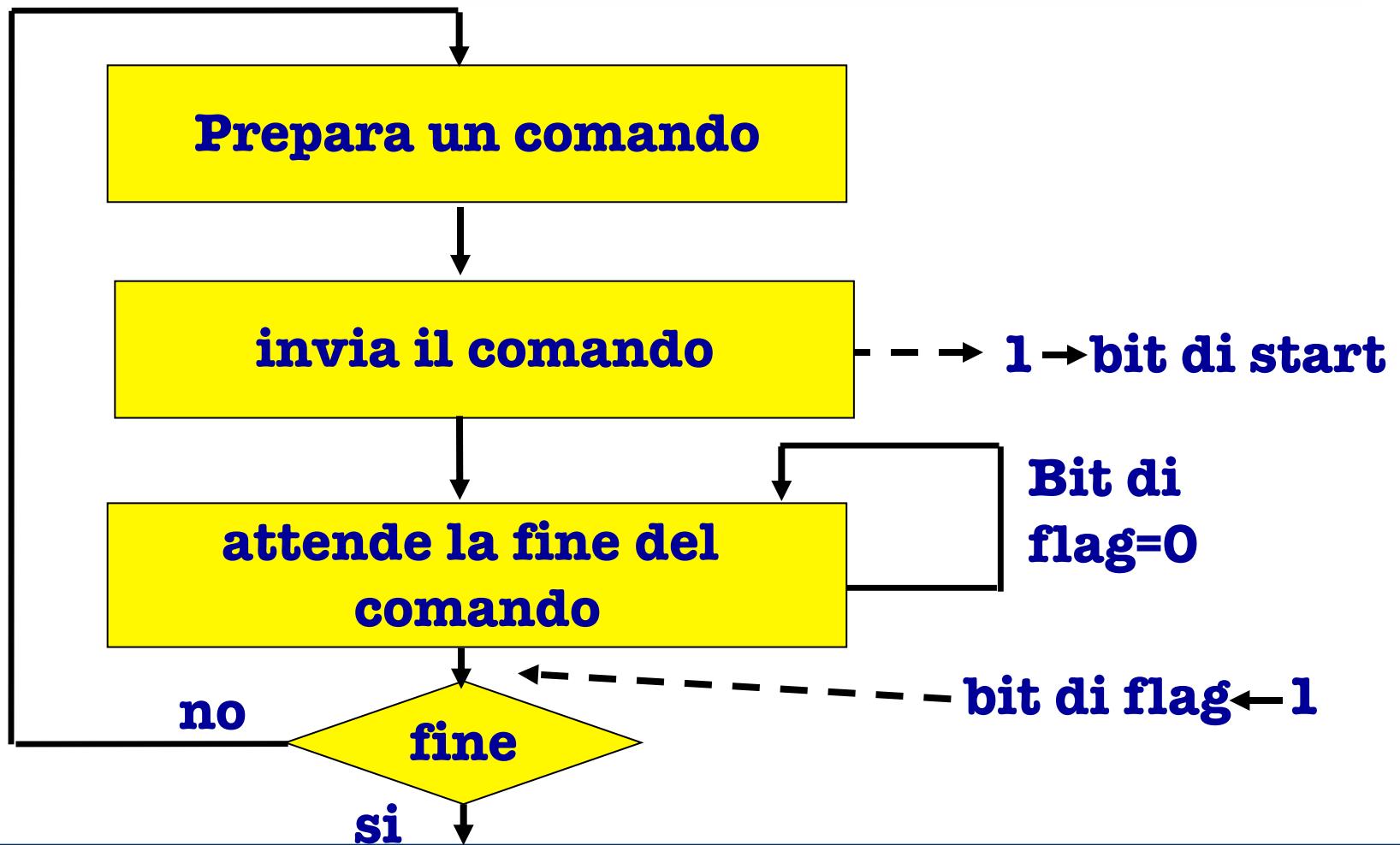
Attività del dispositivo: PROCESSO ESTERNO



Gestione a controllo di programma: PROCESSO ESTERNO

```
processo esterno // descrive l'attività del dispositivo
{
    while (true)
    {
        do{};} while (start ==0)//stand-by
        <esegue il comando>;
        <registra l'esito del comando
            ponendo flag = 1>;
    }
}
```

Gestione a controllo di programma: PROCESSO APPLICATIVO (sulla CPU)



Gestione a controllo di programma: PROCESSO APPLICATIVO:

processo applicativo

{

.....

```
for (int i=0; i++; i<n)
{
    <prepara il comando>;
    <invia il comando>;
    do{} while (flag ==0)
        //ciclo di attesa attiva
    <verifica l' esito>;
}
```

.....

attesa
attiva

GESTIONE basata su INTERRUZIONI

- Lo schema “*a controllo di programma*” non è adatto per sistemi multiprogrammati, a causa dei cicli di attesa attiva.
- Per evitare l’ attesa attiva:
 - Riservare, per ogni dispositivo un semaforo:
semaphore dato_disponibile=0;
Attivare il dispositivo abilitandolo a interrompere
(ponendo nel registro di controllo il bit di
abilitazione a 1).

GESTIONE A INTERRUZIONI

processo applicativo

{

```
.....  
for (int i=0; i++; i<n)  
{    <prepara il comando>;  
    <invia il comando>;  
    p(dato_disponibile) ;  
    <verifica l'esito>;  
}
```

}

.....

cambio
di contesto

FUNZIONE DI RISPOSTA ALLE INTERRUZIONI

```
Interrupt_handler
```

```
{
```

```
.....
```

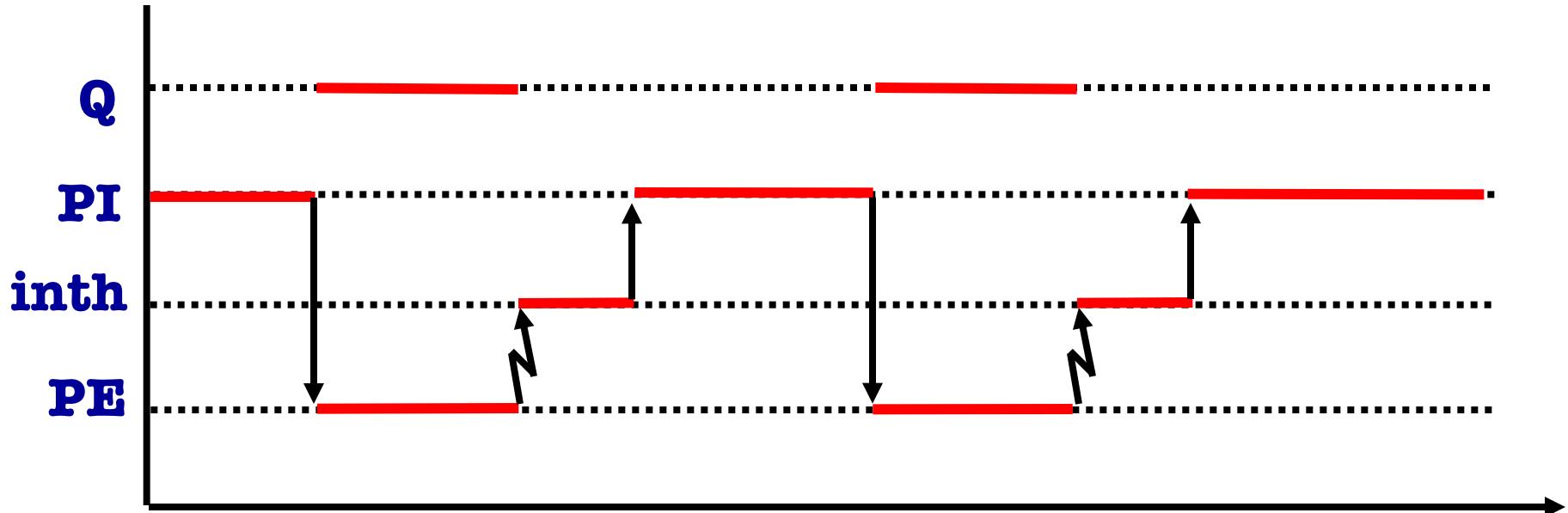
```
v(dato_disponibile) ;
```

```
.....
```

```
}
```

riattiva il
processo
applicativo

DIAGRAMMA TEMPORALE



PI: processo applicativo (“interno”) che attiva il dispositivo

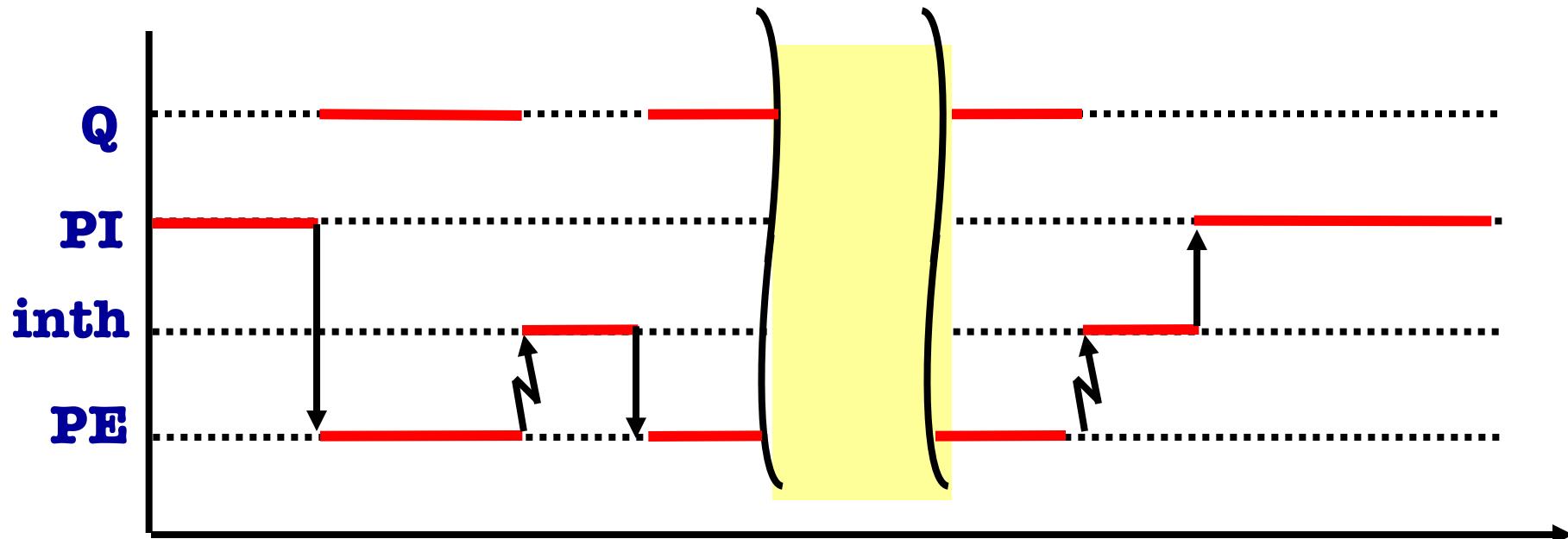
PE: processo esterno

Inth: routine di gestione interruzioni

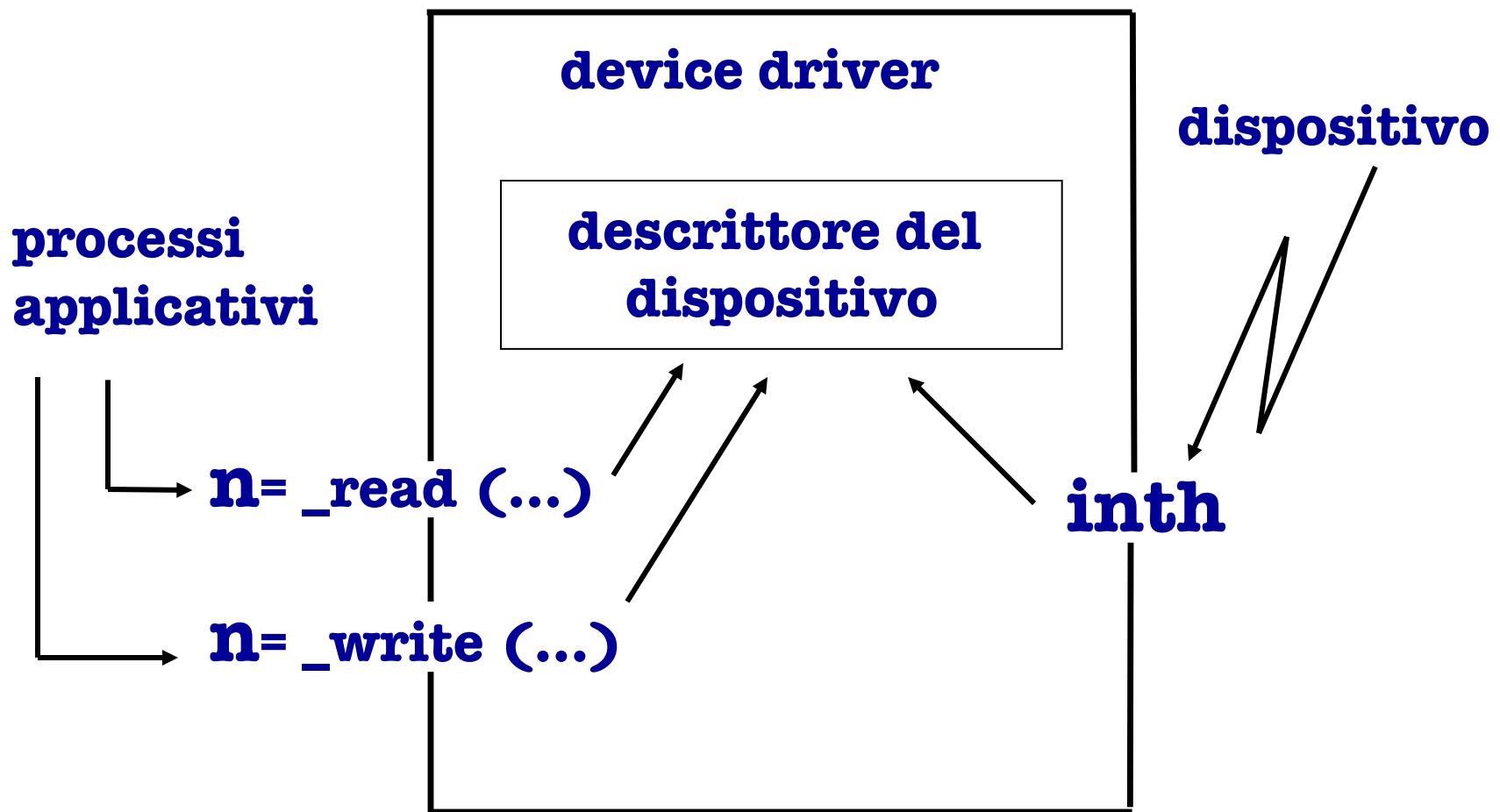
Q: altri processi applicativi

DIAGRAMMA TEMPORALE

E` preferibile uno schema in cui il processo applicativo (PI) che ha attivato un dispositivo per trasferire n dati venga risvegliato solo alla fine dell' intero trasferimento:



ASTRAZIONE DI UN DISPOSITIVO



DESCRITTORE DI UN DISPOSITIVO

indirizzo registro di controllo

indirizzo registro di stato

indirizzo registro dati

semaforo

dato_disponibile

contatore

dati da trasferire

puntatore

al buffer in memoria

esito del trasferimento

DRIVER DI UN DISPOSITIVO

ESEMPIO:

int _read(int disp, char *buf,int cont)

dove:

- la funzione restituisce -1 in caso di errore o il numero di caratteri letti se tutto va bene,
- **disp** è il nome unico del dispositivo,
- **buf** è l' indirizzo del buffer in memoria,
- **cont** il numero di dati da leggere

DRIVER DI UN DISPOSITIVO

```
int _read(int disp,char *buf,int cont)
{ descrittore[disp].contatore=cont;
descrittore[disp].puntatore=buf;
<attivazione dispositivo> ;
p(descrittore[disp].dato_disponibile);
if (descrittore[disp].esito== <cod.errore>)
    return (-1);
return (cont-descrittore[disp].contatore);
}
```

DRIVER DI UN DISPOSITIVO

```
void int() //interrupt handler
{ char b;
<legge il valore del registro di stato>;
if (<bit di errore> == 0)
    {<ramo normale della funzione> }
else
    {<ramo eccezionale della funzione> }
return //ritorno da interruzione
}
```

RAMO NORMALE DELLA FUNZIONE

```
{ < b = registro dati >;  
  * (descrittore[disp].puntatore) = b;  
  descrittore[disp].puntatore ++;  
  descrittore[disp].contatore --;  
  if (descrittore[disp].contatore!=0)  
    <riattivazione dispositivo>;  
 else  
  {descrittore[disp].esito =  
   <codice di terminazione corretta>;  
   <disattivazione dispositivo>;  
   v(descrittore[disp].dato_disponibile);  
 }  
 }
```

RAMO ECCEZIONALE DELLA FUNZIONE

```
{  < routine di gestione errore >;
if (<errore non recuperabile>)
{descrittore[disp].esito = <codice errore>;
v(descrittore[disp].dato_disponibile);
}
}
```

Flusso di controllo durante un trasferimento

!"#\$%&'((*)*'++),(('#)- '

```
Process PI {
    int n;
    int ubufsize = 64;
    char ubuf[ubufsize];
    .....
    .....
    .....
    n=read(IN, ubuf, ubufsize);
    .....
    .....
}
```

Sistema Operativo

```
!"#$%&'((())!'#$%"$!'!"("'!'"')
int read (device dp, char *punt, int cont){
    int n, D;
    char buffer[N];
    < individuazione del dispositivo D coinvolto (naming)>;
    < controllo degli accessi>;
    n = _read(D, buffer, N);
    <trasferimento dei dati da buffer di sistema a ubuf>;
    return n; // ritorno da int.
```

}
②

!"#\$%&'((!)!'#\$%"\$!'!"("'!'"')

```
int _read (int disp, char *pbuf, int cont){
    <attivazione del dispositivo>;
    <sospensione del processo>;
    return (numero dati letti);
```

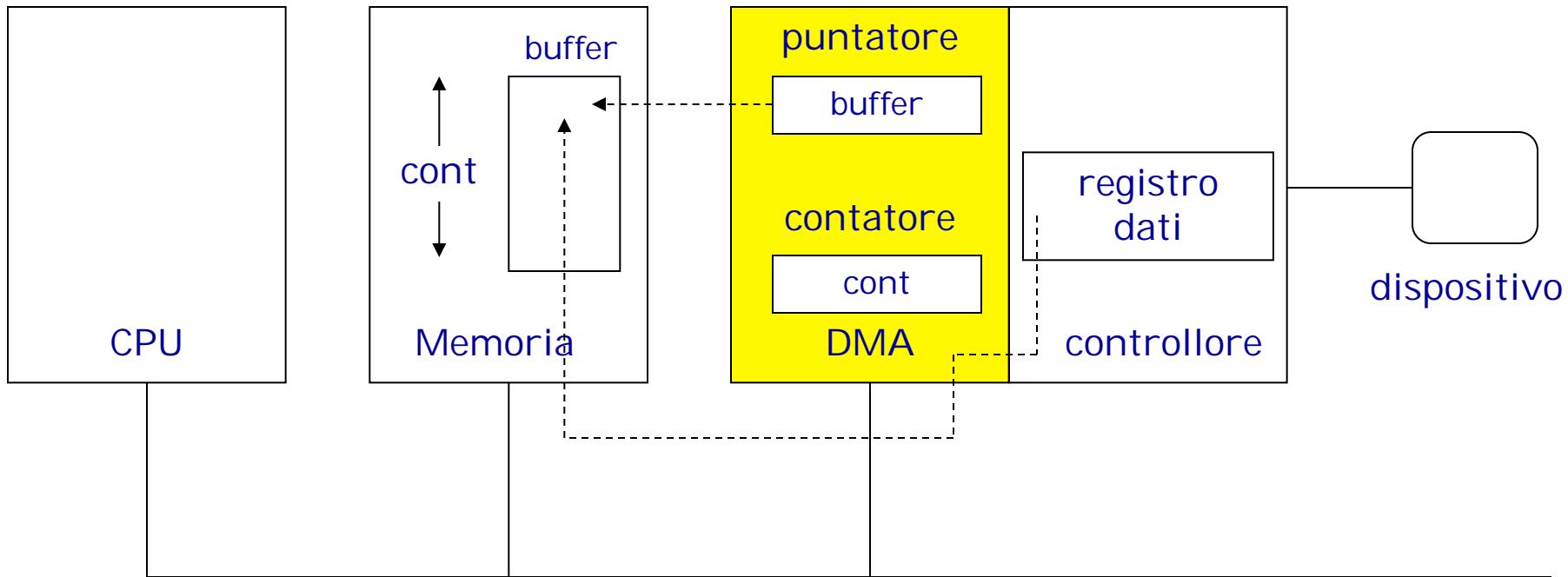
}
④

```
void intch() {
    <trasferimento dati in buffer>;
    <riattivazione processo>
}
```

}
③

hardware

Gestione di un dispositivo in DMA



Gestione di dispositivi speciali : il temporizzatore

- Per consentire la modalità di servizio a divisione di tempo è necessario che il nucleo gestisca un **dispositivo temporizzatore** (timer o clock).
- **Gestione del clock:** i dispositivi clock generano interruzioni periodiche (clock ticks) a frequenze stabilitate; la gestione software delle interruzioni consente di ottenere alcuni servizi quali:
 - aggiornamento della data
 - gestione del quanto di tempo (sistemi time-sharing)
 - valutazione dell'impegno della CPU di un processo
 - gestione della system call ALARM
 - gestione del time-out (watchdog timers)
- Esempio di operazione sul clock: impostazione di un timeout
delay(time);

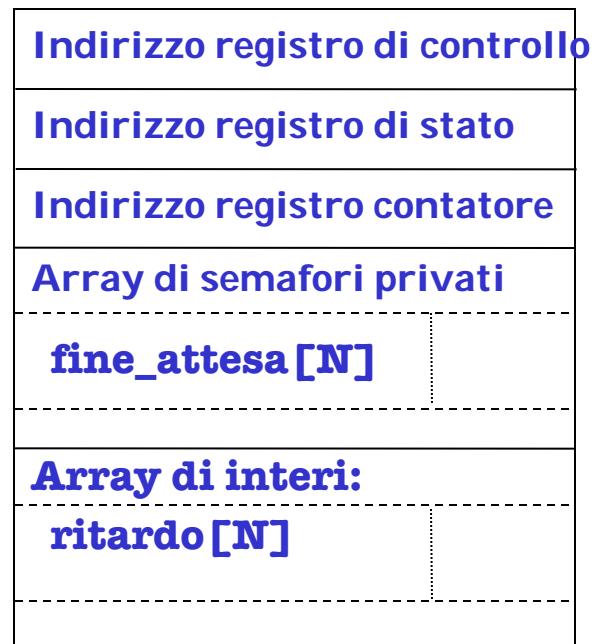
Il controllore del timer contiene, oltre ai registri di controllo e di stato, un **registro contatore** nel quale la CPU trasferisce un valore intero che viene decrementato dal timer.

Quando il registro contatore raggiunge il valore zero il controllore lancia un segnale di interruzione.

Nel descrittore della periferica timer sono presenti:

- un **array di N semafori** (`fine_attesa[N]`) inizializzati a zero. Ciascun semaforo viene utilizzato per bloccare il corrispondente processo che chiama la delay.
- un **array di interi** (`ritardo[N]`) utilizzato per mantenere aggiornato il numero di quanti di tempo che devono ancora passare prima che un processo possa essere riattivato

Descrittore del timer

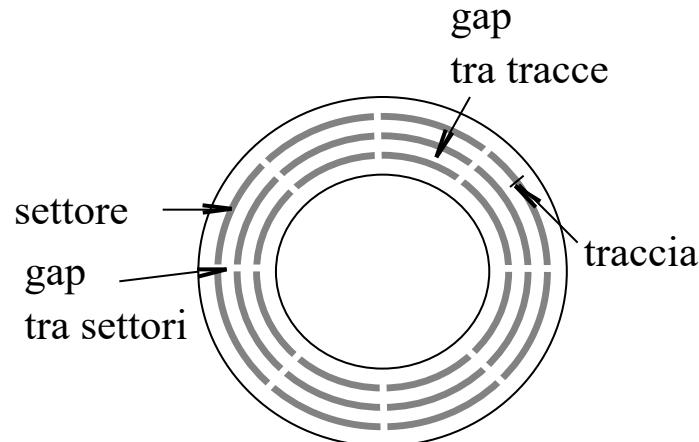


```
void delay (int n) {
    int proc;
    proc=<indice del processo in esecuzione>;
    descrittore.ritardo[proc]= n;
    //sospensione del processo:
    descrittore.fine_attesa[proc].p();
}
```

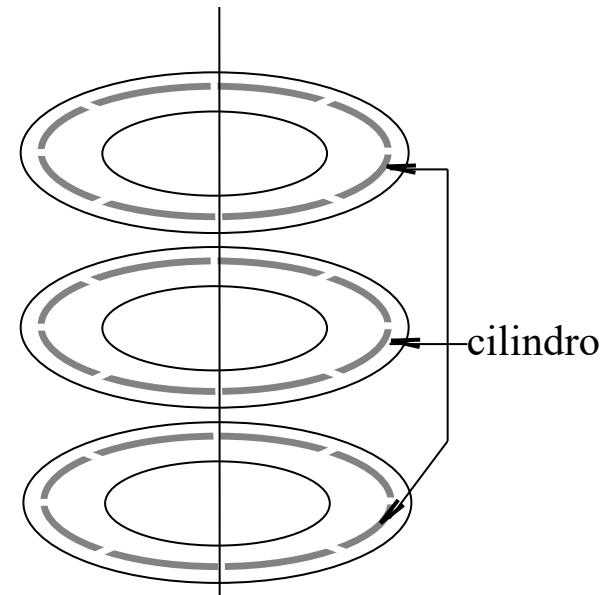
```
void intth(){
    for(int i=0; i<N, i++)
        if (descrittore.ritardo[i]!=0){
            descrittore.ritardo [i]--;
            if (descrittore.ritardo[i]==0)
                descrittore.fine_attesa[i].v();
        }
}
```

Gestione e organizzazione dei dischi

Organizzazione fisica



disco singolo



disk pack

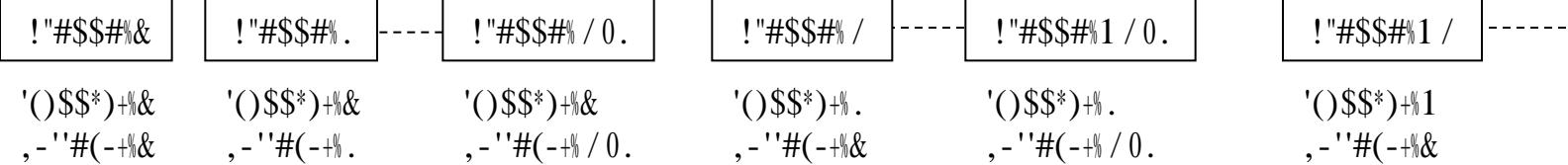
Indirizzo di un settore (blocco fisico):

(f,t,s)

dove:

f numero della **faccia**, t numero della **traccia**
nell'ambito della faccia, s numero del **settore** all'interno
della faccia.

Tutti i **blocchi** che compongono un disco (o un pacco di dischi), possono essere trattati come un array lineare di blocchi.



Indicando con

M il numero di tracce per faccia

N numero di settori per traccia

un **blocco** di coordinate (f,t,s) viene rappresentato
nell'ambito dell'array con l'indice i

$$i = f * M * N + t * N + s$$

Scheduling delle richieste di trasferimento

$$\mathbf{TF} = \mathbf{TA} + \mathbf{TT}$$

- TF** tempo medio di trasferimento di un blocco (per leggere o scrivere un blocco)
- TA** tempo medio di accesso (per posizionare la testina di lettura/ scrittura all'inizio del blocco considerato)
- TT** tempo di trasferimento dei dati del blocco

$$\mathbf{TA} = \mathbf{ST} + \mathbf{RL}$$

- ST** tempo di *seek* (per posizionare la testina sopra la traccia contenente il blocco considerato)
- RL** *rotational latency*: tempo necessario perché il settore, ruotando, si posizioni sotto la testina)

Parametri	AC2540	WDE18300
Numero cilindri (N. di tracce per ogni faccia)	1048	13614
Tracce per cilindro	4	8
Settori per traccia	252	320
Byte per settore	512	512
Capacità	540 MB	18.3 GB
Tempo minimo di seek (tra cilindri adiacenti)	4 msec.	0.6 msec.
Tempo medio di seek	11 msec.	5.2 msec.
Tempo di rotazione	13 msec.	6 msec.
Tempo di trasferimento di un settore	53 μ s	19 μ s

Tabella 5.2 parametri caratterizzanti i due dischi WD AC2540 e WDE18300.

TT tempo necessario per far transitare sotto la testina **l'intero blocco**. Indicando con **t** il tempo necessario per compiere un giro, s il numero di settori per traccia, si ha

$$\mathbf{TT} = \mathbf{t/s} \text{ (valore approssimato, } \mu\text{s}).$$

Quindi:

$$\mathbf{TF = ST + RL + TT}$$

The diagram shows the formula $TF = ST + RL + TT$ with a bracket underneath the terms ST , RL , and TT , indicating they are summed together to form the total transfer time TA .

Il tempo medio di trasferimento dipende sostanzialmente dal tempo medio di accesso ($TA=ST + RL$):

- RL è un parametro che dipende dalle caratteristiche fisiche del dispositivo
- ST dipende da come il disco viene gestito -> **possibilità di gestione mirata alla riduzione del valore di ST**

Due modi di intervento:

- Criteri con cui i *blocchi sono memorizzati* su disco
(metodo di **allocazione** dei file)
- Criteri con cui *servire le richieste* di accesso
(politiche di **scheduling** delle richieste)

Politiche di Scheduling delle Richieste

Le richieste in coda ad un dispositivo possono essere servite secondo diverse politiche:

- First-Come-First-Served (FCFS)
- Shortest-Seek-Time-First (SSTF)
- SCAN algorithm
- C-SCAN (Circular-SCAN)

FCFS. Le richieste sono servite rispettando il tempo di arrivo. Si evita il problema della *starvation*, ma non risponde ad alcun criterio di ottimalità.

SSTF. Seleziona la richiesta con **tempo di seek minimo** a partire dalla posizione attuale della testina; può provocare situazioni di *starvation*

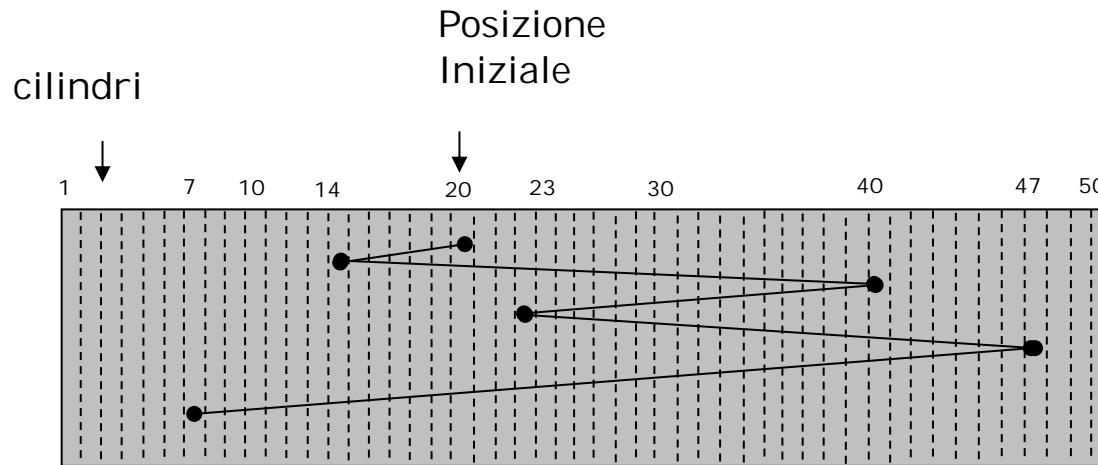
SCAN. La testina si porta ad una estremità del disco e si sposta verso l'altra estremità, servendo le richieste man mano che vengono raggiunte le tracce indicate, fino all'altra estremità del disco. Quindi viene invertita la direzione.

-> **Minimizzazione del ST medio**

CSCAN: Nella valutazione del tempo medio di attesa di un **processo**, e` **necessario** tenere in conto anche il tempo durante il quale il processo attende che la sua richiesta di accesso venga servita: la politica **CSCAN** è una variante dello SCAN mirata a fornire un tempo di attesa medio (dei processi) più basso.

Algoritmo di scheduling FCFS

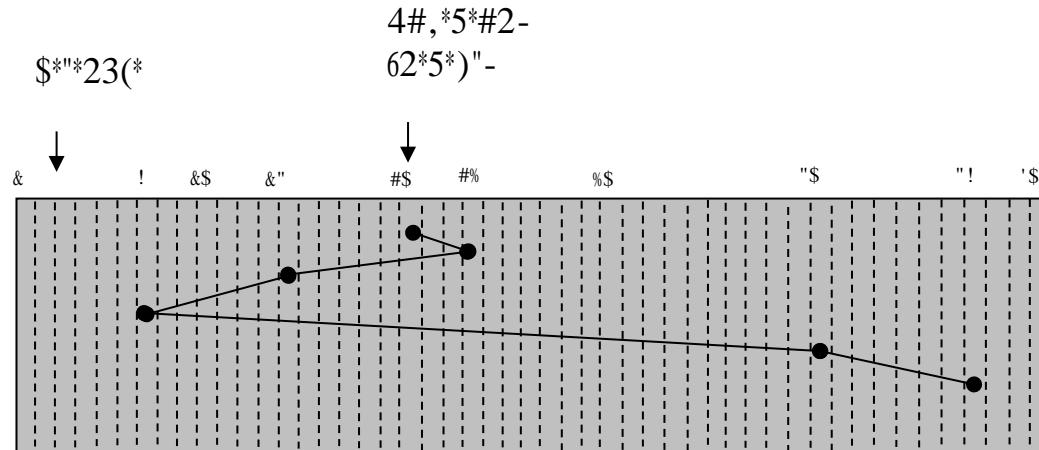
Esempio: Testina posizionata sul cilindro 20.
Richieste presenti in coda: [14, 40, 23, 47, 7]



Spostamento totale = 113 cilindri

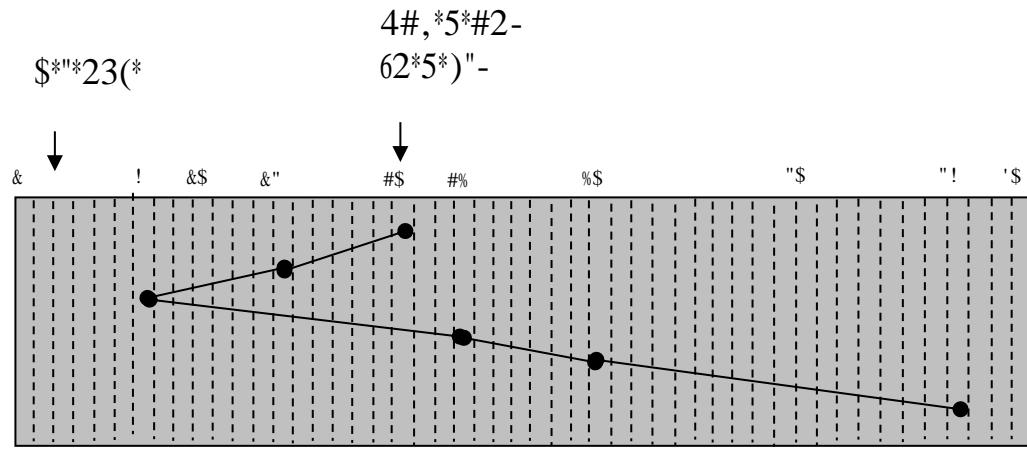
Algoritmo di Scheduling SSTF

Testina posizionata sul cilindro 20. Richieste presenti in coda: [14, 40, 23, 47, 7]



Spostamento totale = 59 cilindri

Algoritmo di Scheduling SCAN



Spostamento totale = 53 cilindri

C-SCAN (circular scan)

L'algoritmo SCAN non considera il tempo di attesa dei processi. Quando viene invertita la direzione per iniziare una nuova scansione, le richieste più “vecchie” riguardano tracce più lontane, che quindi verranno servite alla fine della scansione.

CSCAN è una variante dello SCAN, nella quale l'insieme delle tracce viene scansionato sempre nella stessa direzione (non c'è inversione di direzione): arrivata all'ultima traccia, la testina ritorna immediatamente all'inizio del disco per una nuova scansione a partire dalla prima traccia. Rispetto allo SCAN, fornisce un tempo di attesa medio più basso.

Quinta Esercitazione

Comunicazione tra
processi Unix: pipe

System Call relative alle pipe

pipe	<ul style="list-style-type: none">• <code>int pipe (int fd[])</code> crea una pipe e assegna i 2 file descriptor relativi agli estremi di lettura/scrittura ai primi due elementi dell'array fd.• Restituisce 0 in caso di creazione con successo, -1 in caso di errore
close	<ul style="list-style-type: none">• Stessa system call usata per chiudere file descriptor di file regolari• Nel caso di pipe, usata da un processo per chiudere l'estremità della pipe che non usa.

Primitive di comunicazione

read	<ul style="list-style-type: none">Stessa system call usata per leggere file regolari, ma può essere bloccante: Se la pipe è vuota: il processo chiamante attende fino a quando non ci sono dati disponibili.
write	<ul style="list-style-type: none">Stessa system call usata per scrivere su file regolari, ma può essere bloccante: Se la pipe è piena: il processo chiamante attende fino a quando non c'è spazio sufficiente per scrivere il messaggio.
dup	<p>fd1=dup(fd) crea una copia dell'elemento della tabella dei file aperti di indice fd.</p> <ul style="list-style-type: none">La copia viene messa nella prima posizione libera (in ordine crescente di indice) della tabella dei file aperti.Assegna a fd1 l'indice della nuova copia, -1 in caso di errore

Esempio – comunicazione tra processi mediante pipe

Realizzare un programma C che, utilizzando le *system call* di UNIX, preveda un'interfaccia del tipo:

esame F1 F2.. FN PAROLA

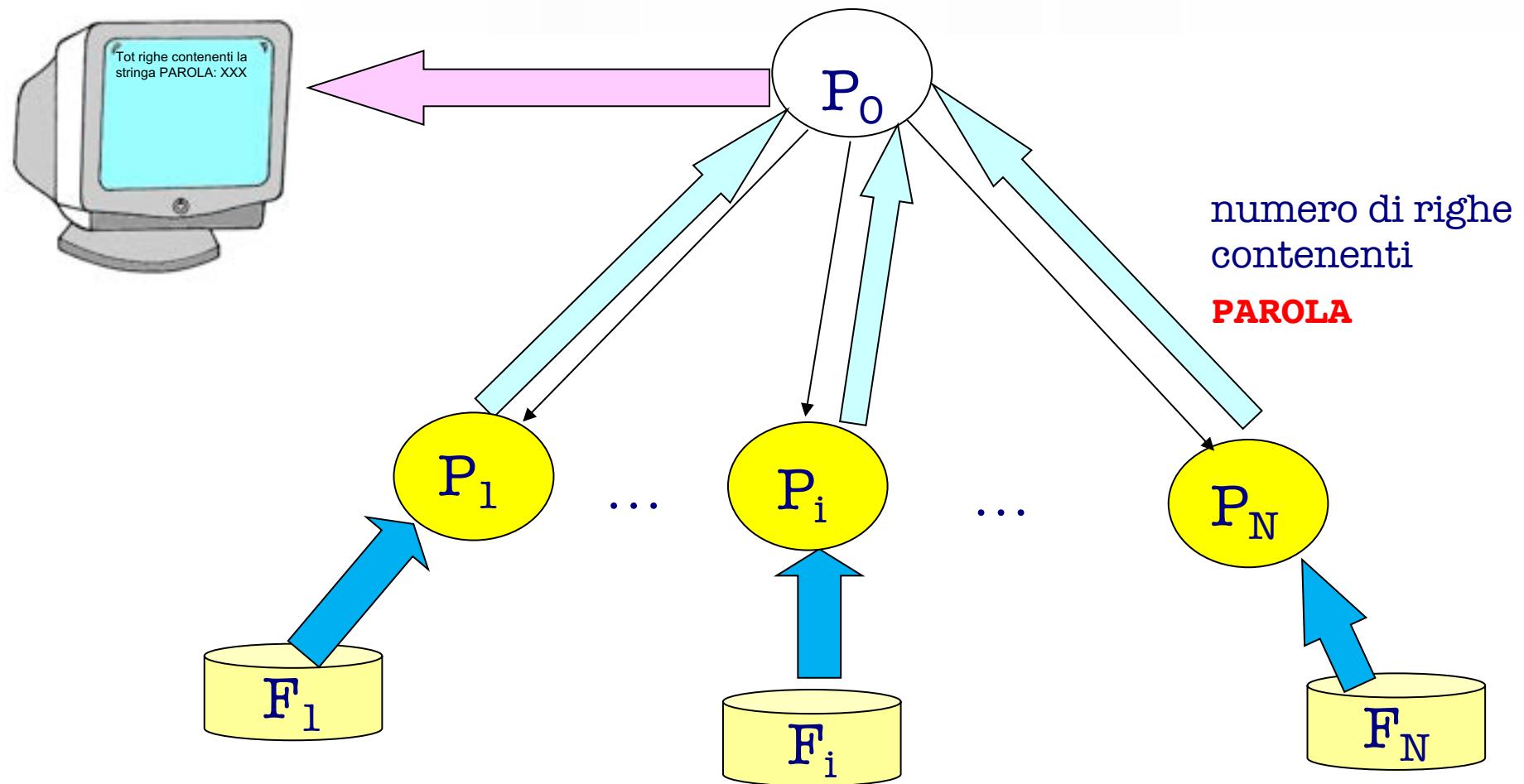
- **F1, F2,.. FN** rappresentano nomi assoluti di file
- **PAROLA** rappresenta una stringa

Il processo padre **P0** genera N figli (tanti quanti sono i filename dati come argomenti) P1, P2, .. PN:

Ogni figlio P_i , **tramite il comando grep**, deve **contare** le righe di file_i nelle quali la stringa **PAROLA** compare almeno una volta, **e comunicare a P0 il valore risultante da tale conteggio**.

Il padre P0, sommerà i risultati ricevuti da tutti i figli, **stamperà sullo standard output il numero totale di righe contenenti almeno un'occorrenza** di **PAROLA**, e successivamente terminerà.

Esempio - Modello di soluzione



Esempio - Considerazioni

Ogni figlio Pi deve contare tramite l'esecuzione del comando **grep**. Per ottenere il numero di righe contenenti la stringa PAROLA, usare l'opzione **-c** (v. man grep).

Esempio di invocazione:

```
$ grep -c PAROLA FileX
```

Attenzione:

L'output di **grep -c** è una **stringa** che rappresenta il numero di occorrenze di **C**. **Non è di tipo intero!**

Comunicazione figli->padre: quante pipe?

Il padre non ha bisogno distinguere il mittente di ogni messaggio ricevuto: il suo compito è sommare tutti i valori ricevuti → usiamo **1 pipe**.

Per ogni processo, l'output di grep deve essere inviato al padre PO → ridirezione su pipe.

Soluzione dell'esercizio

```
#include <fcntl.h>
#include <stdio.h>
...
#define NP 8 // al massimo 8 figli
#define MAXS 256

void figlio(int fd_out, char filein[], char word[]);
void wait_child();

int pp[2]; // pipe per la comunicazione figli ->padre
```

```
int main(int argc, char **argv) {
    int pid[NP];
    int i,N; // numero di figli
    char parola[MAXS],buffer[MAXS];
    int tot_occ=0;
    // controllo argomenti:
    if (argc<3) // almeno 2 argomenti F1.. FN parola
    {   printf("sintassi! %s F1.. FN parola\n", argv[0]);
        exit(-1);
    }
    N=argc-2; // N è il numero di file -> numero dei figli
    strcpy(parola, argv[argc-1]);
    if (N>NP)
    {   printf("troppi file !\n");
        exit(-2);
    }
```

```
// Apertura pipe pp:  
if (pipe(pp)<0)  
    exit(-3); /* apertura pipe fallita */  
  
/* creazione figli: */  
for(i=0;i<N;i++)  
{  if ((pid[i]=fork())<0)  
    {      perror("fork error");  
        exit(-3);  
    }  
    else if (pid[i]==0) // figlio i  
    {      close(pp[0]); //chiude il lato di lettura di pp  
        figlio(pp[1], argv[i+1], parola);  
    }  
}
```

```
// Padre:  
close(pp[1]); // chiude il lato di scrittura di pp  
  
for(i = 0; i < N; i++)  
{    char c;  
    int letto, cont, fine,nread;  
cont=0;  
fine=0;  
while(!fine)  
{    nread=read(pp[0], &c, 1); // leggo il prossimo char dalla pipe  
    if ((nread==1)&&(c!='\n')) // ho letto un char significativo  
    {        buffer[cont] = c;  
        cont++;  
    }  
    else {  
        fine=1;  
    }  
}  
buffer[cont]='\0';  
letto=atoi(buffer);  
tot_occ+=letto;  
}
```

```

sprintf(buffer,"%d\n", tot_occ);
write(1,buffer,strlen(buffer));// stampa il risultato
// attesa figli:
for(i = 0; i < N; i++)
{   wait_child();
}
close(pp[0]); // chiudo la pipe
exit(0);
} /* fine main */

void figlio(int fd_out, char filein[], char word[])
{ // ridirezione output su lato scrittura pipe:
close(1);
dup(fd_out);
close(fd_out);
// esecuzione grep:
execl("/bin/grep", "grep", "-c", word, filein, (char*)0);
perror("Execl fallita");
exit(-1);
}

```

```
void wait_child() {
    int pid_terminated, status;
    pid_terminated=wait(&status);
    if(WIFEXITED(status))
        printf("PADRE: terminazione volontaria del figlio %d con stato
%d\n",
               pid_terminated, WEXITSTATUS(status));
    else if(WIFSIGNALED(status))
        printf("PADRE: terminazione involontaria del figlio %d a causa
del segnale %d\n",
               pid_terminated,WTERMSIG(status));
}
```

Sistemi Operativi T

Processi e Thread

Concetto di processo

Il processo è un
programma in esecuzione

- È l'**unità di esecuzione** all'interno del SO
 - Solitamente, esecuzione **sequenziale** (istruzioni vengono eseguite in sequenza, secondo l'ordine specificato nel testo del programma)
 - un SO multiprogrammato consente l'esecuzione **concorrente** di più processi
- 👉 D'ora in poi faremo implicitamente riferimento sempre al caso di SO multiprogrammati

Concetto di processo

Programma = entità passiva
Processo = entità attiva

Il processo è rappresentato da:

- **codice** (text) del programma eseguito
- **dati**: variabili globali
- **program counter**
- altri **registri** di CPU
- **stack**: parametri, variabili locali a funzioni/procedure

Inoltre, a un processo possono essere associate delle **risorse**.

Ad esempio:

- file aperti
- connessioni di rete utilizzate
- altri dispositivi di I/O in uso
- ...

Processo = {PC, registri, stack, text, dati, risorse}

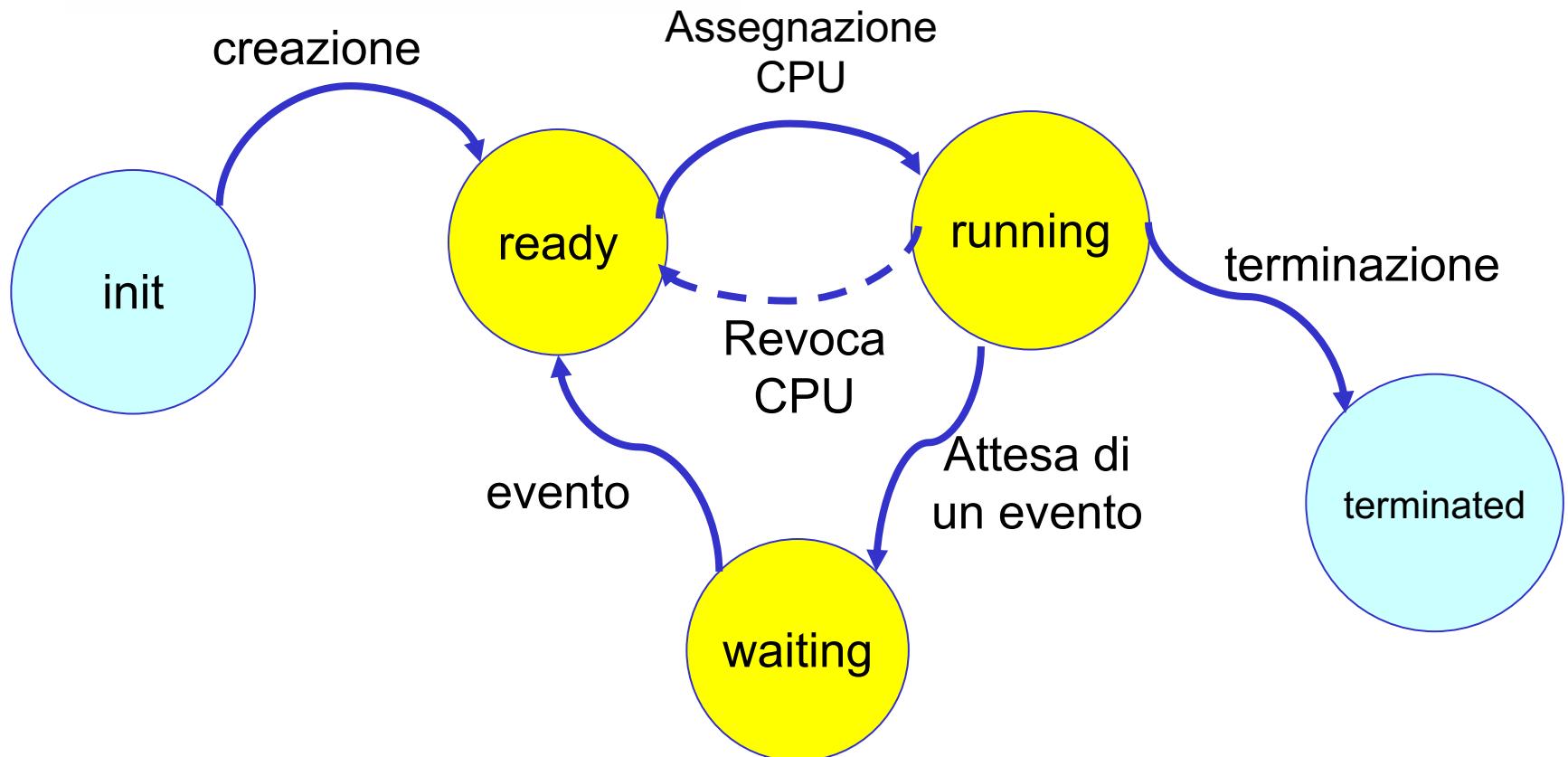
Stati di un processo

Un processo, durante la sua esistenza può trovarsi in vari **stati**:

- **Init**: stato transitorio durante il quale il processo viene caricato in memoria e SO inizializza i dati che lo rappresentano
- **Ready**: processo è pronto per acquisire la CPU
- **Running**: processo sta utilizzando la CPU
- **Waiting**: processo è sospeso in attesa di un evento
- **Terminated**: stato transitorio relativo alla fase di terminazione e deallocazione del processo dalla memoria

Stati di un processo

Transizioni di stato:



Stati di un processo

Transizioni di stato:



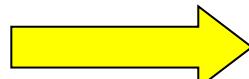
Stati di un processo

In un sistema **monoprocesso** e **multiprogrammato**:

- un solo processo (al massimo) si trova nello stato running
- più processi possono trovarsi contemporaneamente negli stati ready e waiting

Necessità di strutture dati per mantenere in memoria le informazioni su processi in attesa:

- di acquisire la CPU (ready)
- di eventi (waiting)



**Descrittore di processo
(o Process Control Block)**

Rappresentazione dei processi

Come vengono rappresentati i processi nel S.O.?

- Ad ogni processo viene associata una struttura dati (**descrittore**): **Process Control Block (PCB)**
- il **PCB** contiene tutte le informazioni relative al processo:
 - **Stato del processo**: ready, running, waiting, ecc.
 - Contenuto dei **registri** di CPU (PC, SP, IR, accumulatori, ...)
 - Informazioni di **scheduling** (priorità, puntatori alle code, ...)
 - Informazioni per gestore di **memoria** (registri base, limite, ...)
 - **Informazioni relative all'I/O** (risorse allocate, file aperti, ...)
 - Informazioni di accounting (tempo di CPU utilizzato, ...)
 - ...

Process Control Block

stato del processo
identificatore del processo
PC
registri
limiti di memoria
file aperti
...

Il sistema operativo gestisce i PCB di tutti i processi, organizzandoli in opportune strutture dati (ad esempio code) a seconda del loro stato.

Scheduling dei processi

È l'attività mediante la quale il SO effettua delle scelte tra i processi, riguardo a:

- caricamento in memoria centrale
- assegnazione della CPU

In generale, il SO compie tre diverse attività di scheduling:

- scheduling a breve termine (o di CPU)
- scheduling a medio termine (o swapping)
- [scheduling a lungo termine]

Scheduler a lungo termine

Lo scheduler a lungo termine è quella componente del SO che seleziona i programmi da eseguire dalla memoria secondaria per caricarli in memoria centrale (creando i corrispondenti processi):

- controlla il grado di multiprogrammazione (numero di processi contemporaneamente presenti nel sistema)
- è una componente importante dei sistemi **batch multiprogrammati**
- nei sistemi **time sharing** non è presente:

Interattività: di solito è l'utente che stabilisce direttamente quali/quanti processi caricare => scheduler a lungo termine non è presente

Scheduler a medio termine (swapper)

Nei sistemi operativi multiprogrammati:

- la quantità di memoria fisica può essere minore della somma delle dimensioni delle aree di memoria da allocare a ciascun processo
- il grado di multiprogrammazione* non è, in generale, vincolato dalle esigenze di spazio dei processi

Swapping: trasferimento temporaneo in memoria secondaria di processi (o di parti di processi), in modo da consentire il caricamento di altri processi

* massimo numero di processi gestibili contemporaneamente dal SO

Scheduler a breve termine (o di CPU)

È quella parte del SO che si occupa della selezione dei processi a cui assegnare la CPU

Nei sistemi multiprogrammati, ogni volta che il processo corrente lascia la CPU (attesa di un evento o, in sistemi time sharing, allo scadere del quanto di tempo) il SO:

- **decide** a quale processo assegnare la CPU (scheduling di CPU)
- **effettua** il cambio di contesto (context switch)

Cambio di contesto

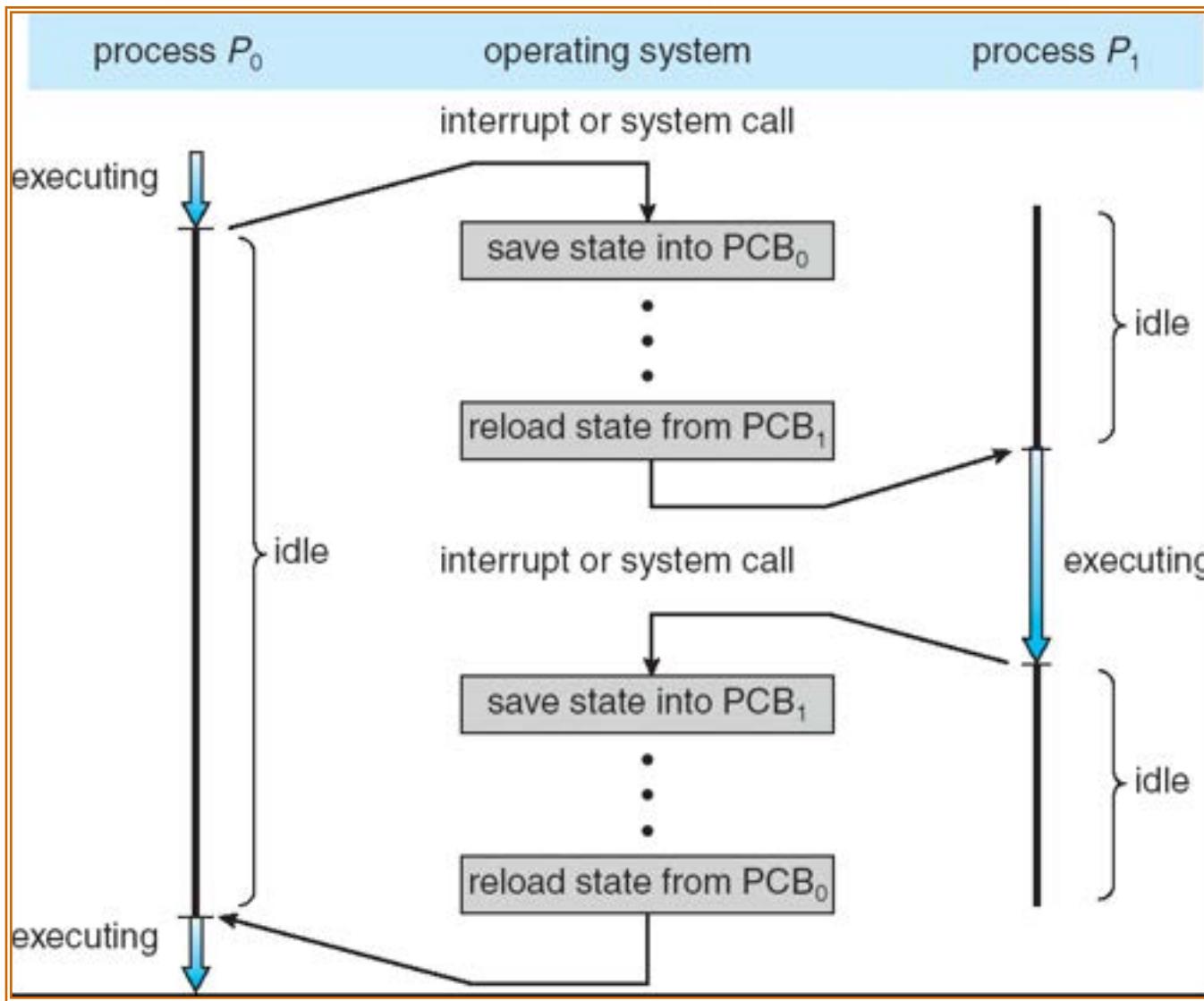
È la fase in cui l'uso della CPU viene commutato da un processo ad un altro

Quando avviene un **cambio di contesto** tra un processo P_i ad un processo P_{i+1} (ovvero, P_i cede l'uso della CPU a P_{i+1}):

- **Salvataggio dello stato di P_i :** SO copia PC, registri, ... del processo deschedulato P_i nel suo PCB
- **Ripristino dello stato di P_{i+1} :** SO trasferisce i dati del processo P_{i+1} dal suo PCB nei registri di CPU, che può così riprendere l'esecuzione

→ Il passaggio da un processo al successivo può richiedere onerosi trasferimenti da/verso la memoria secondaria, per allocare/deallocare gli spazi di indirizzi dei processi (vedi gestione della memoria)

Cambio di contesto



Scheduler a breve termine (o di CPU)

Lo scheduler a breve termine gestisce

- la **coda dei processi pronti**: contiene i PCB dei processi che si trovano in stato *Ready*

Altre strutture dati necessarie:

- **code di waiting** (una per ogni tipo di attesa: dispositivi I/O, timer, ...): **ognuna di esse contiene i PCB dei processi waiting** in attesa di un evento del tipo associato alla coda

Scheduler

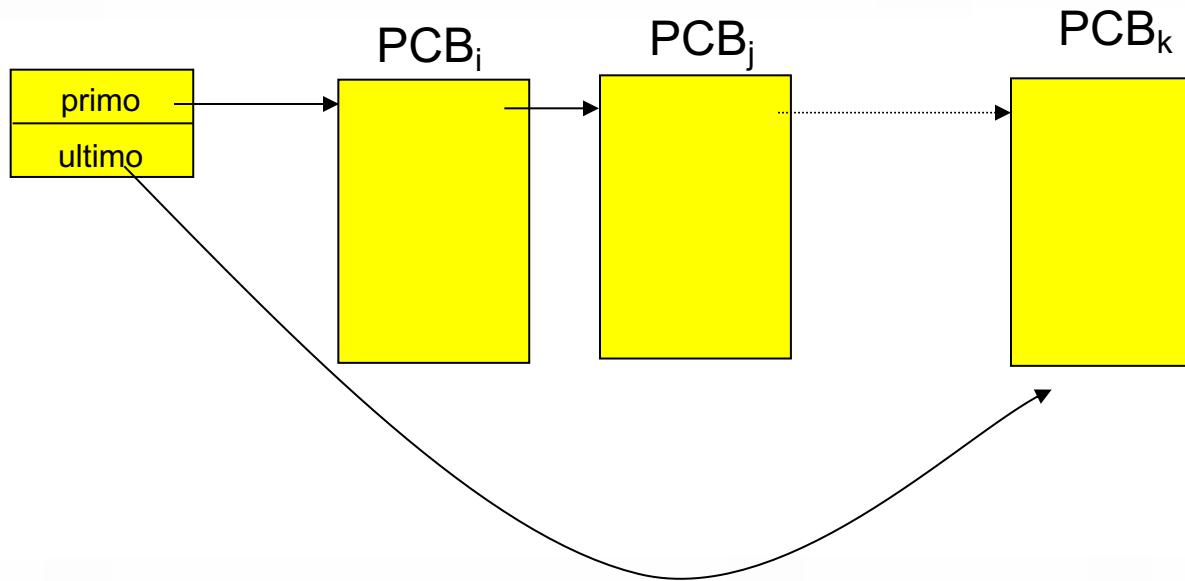
- **Scheduler short-term** viene invocato con alta frequenza (ms) -> deve essere molto efficiente
- **Scheduler medium-term** viene invocato a minore frequenza (sec-min) -> può essere anche più lento

Scelte ottimali di scheduling dipendono dalle caratteristiche dei processi. Ad es.:

- processi I/O-bound – maggior parte del tempo in operazioni I/O
- processi CPU-bound – maggior parte del tempo in uso CPU

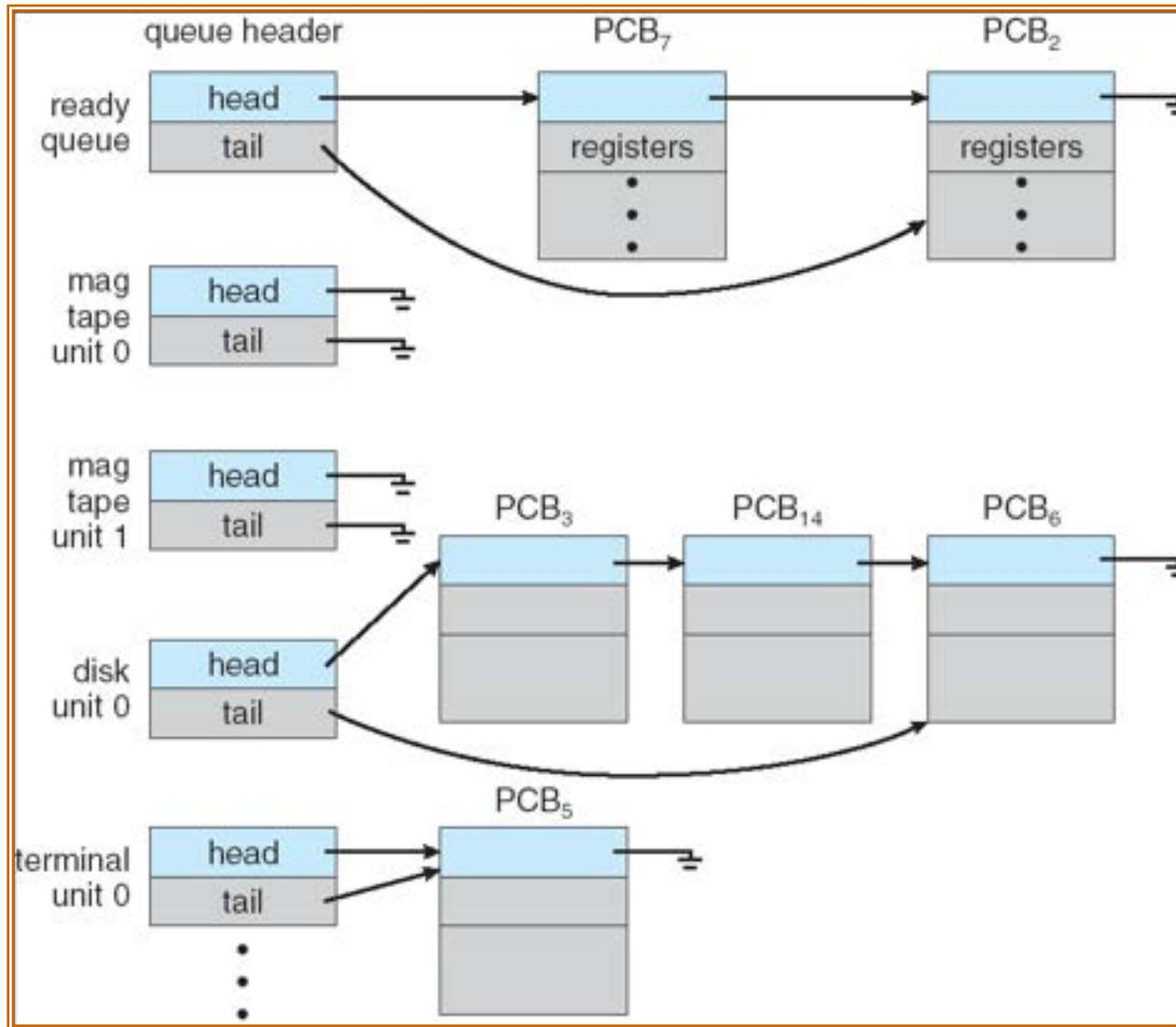
Code di Scheduling

Coda dei processi pronti (*ready queue*):



- ❖ strategia di gestione della ***ready queue*** dipende dalle ***politiche (algoritmi) di scheduling*** adottate dal SO
(parleremo di algoritmi di scheduling nelle prossime lezioni)

Ready queue e altre code per dispositivi I/O



Scheduling e cambio di contesto

Operazioni di scheduling determinano un costo computazionale aggiuntivo (**overhead**) che dipende essenzialmente da:

- frequenza di cambio di contesto
- dimensione PCB
- costo dei trasferimenti da/verso la memoria

➔ esistono SO che prevedono processi leggeri (**thread**) che hanno la proprietà di condividere codice e dati con altri processi:

- dimensione PCB ridotta
- **riduzione overhead**

Operazioni sui processi

Ogni SO multiprogrammato prevede dei meccanismi per la gestione dei processi

Meccanismi necessari:

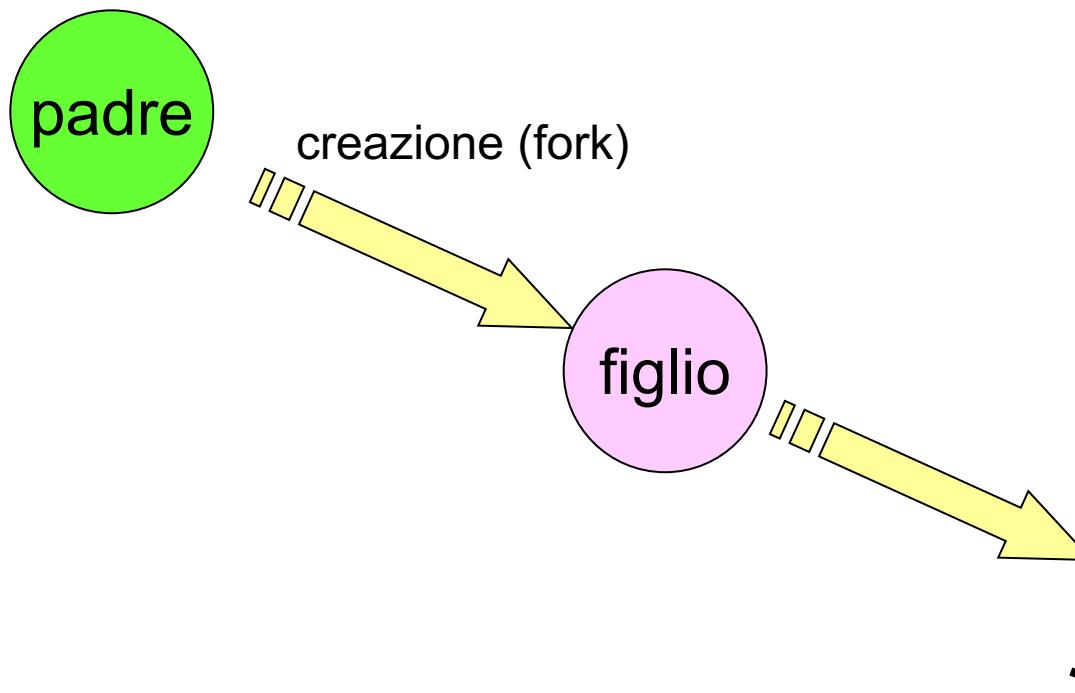
- **creazione**
- **terminazione**
- **interazione** tra processi

Sono operazioni privilegiate (esecuzione in modo kernel)

→ definizione di **system call**

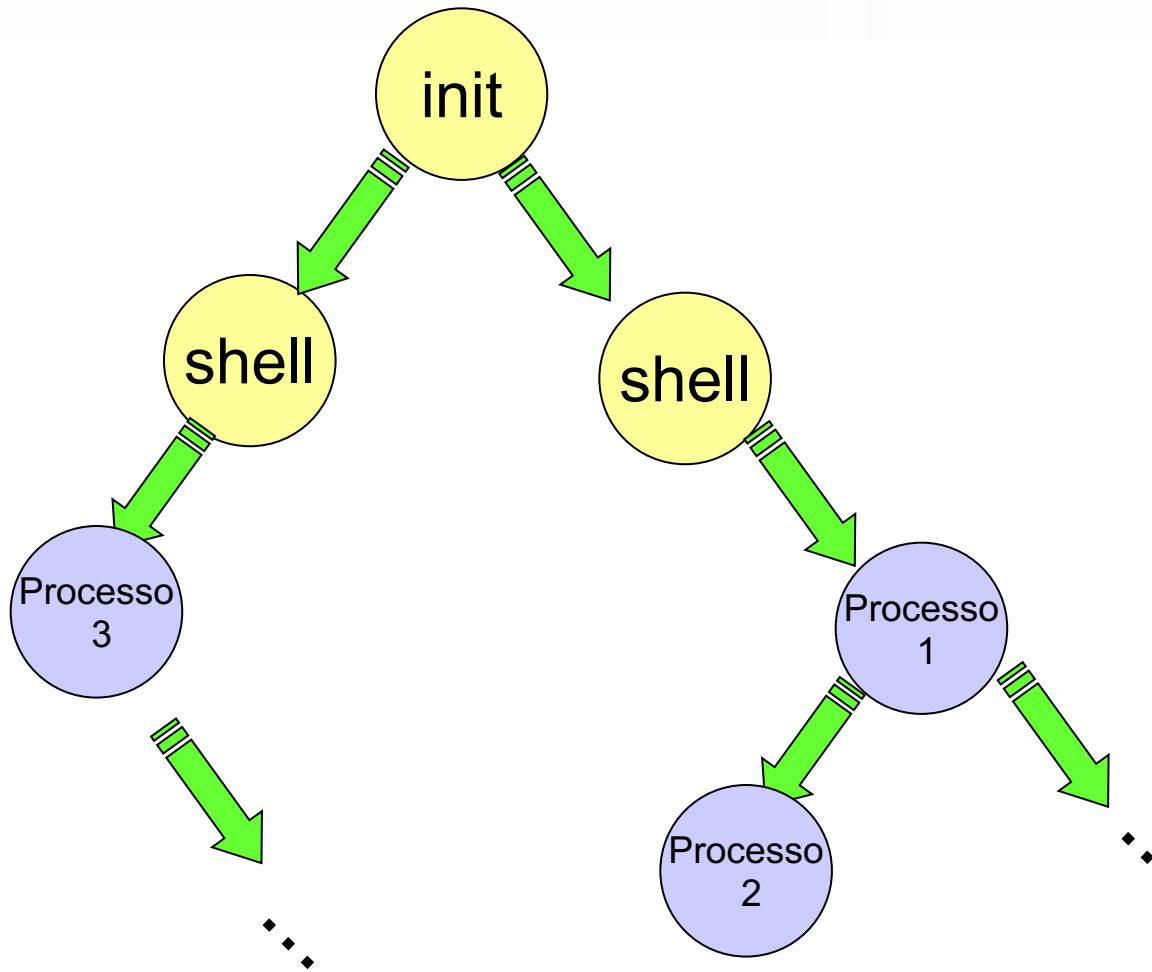
Creazione di processi

- Un processo (**padre**) può richiedere la creazione di un nuovo processo (**figlio**)



- È possibile realizzare gerarchie di processi

Gerarchie di processi (es. UNIX)



Relazione padre-figlio

Aspetti caratteristici:

- **concorrenza**

Ad esempio:

- padre e figlio procedono in parallelo (es. UNIX), oppure
- il padre genera uno o più figli concorrenti e attende la loro terminazione
- ...

- **condivisione di risorse**

Ad esempio:

- le risorse del padre (ad esempio, i file aperti) sono condivise con i figli (es. UNIX), oppure
- il figlio utilizza risorse soltanto se esplicitamente richieste da se stesso
- ...

- **spazio degli indirizzi**

Ad esempio:

- **duplicato**: lo spazio degli indirizzi del figlio è una **copia** di quello del padre : stesso codice, copia degli stessi dati, ecc. (es. fork() in UNIX) oppure
- **differenziato**: spazi degli indirizzi di padre e figlio con codice e dati diversi (es. VMS, stesso processo dopo exec() in UNIX)

Terminazione

Ogni processo:

- è **figlio** di un altro processo
- può essere a sua volta **padre** di processi

Se un processo termina:

- il **padre** può rilevare il suo stato di terminazione
- possibili effetti sui **figli**:
 - tutti i **figli terminano**, oppure
 - i **figli continuano l'esecuzione** e assumono come padre «adottivo» un altro processo (es: Unix)

SO deve mantenere le informazioni relative alle relazioni di **parentela** nel **descrittore** di ogni processo:

- ad esempio, riferimento al **padre**, e/o
- riferimenti agli eventuali **figli**

Processi leggeri (thread)

Un **thread** (o processo leggero) è un'unità di esecuzione che condivide codice e dati con altri thread ad esso associati.

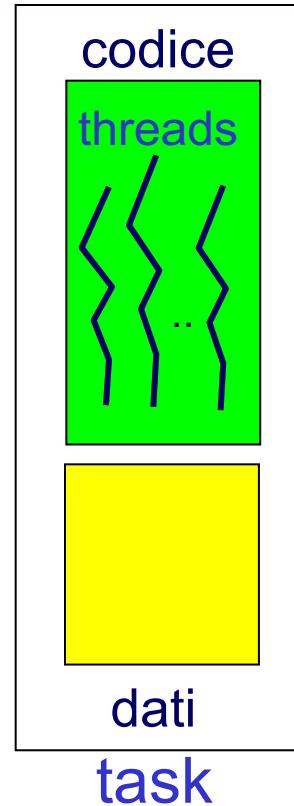
Task = insieme di thread che riferiscono lo stesso codice (text) e gli stessi dati.

- **codice, dati e risorse** non sono caratteristiche del singolo thread, ma del task al quale appartengono
- ogni thread è caratterizzato da un PC, registri e stack privati -> concorrenza

Thread = {PC, registri, stack, ...}

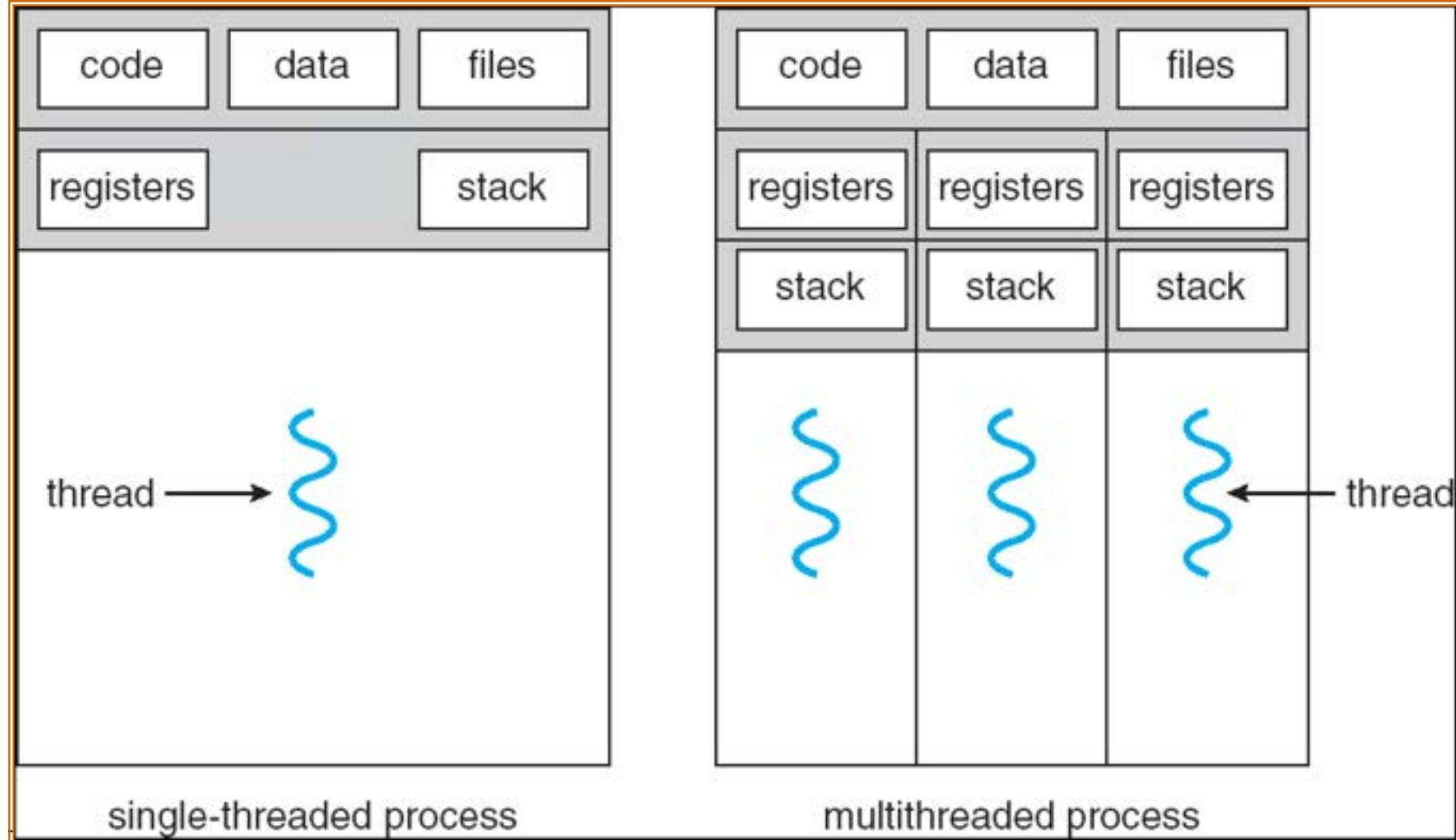
Task = {thread1, thread2, ..., threadN, text, dati, risorse}

Processi leggeri (thread)



- **Processo pesante** equivale a un task con un solo thread (*single-threaded process*)

Processi single- o multi-threaded



single-threaded process

multithreaded process

Vantaggi dei thread

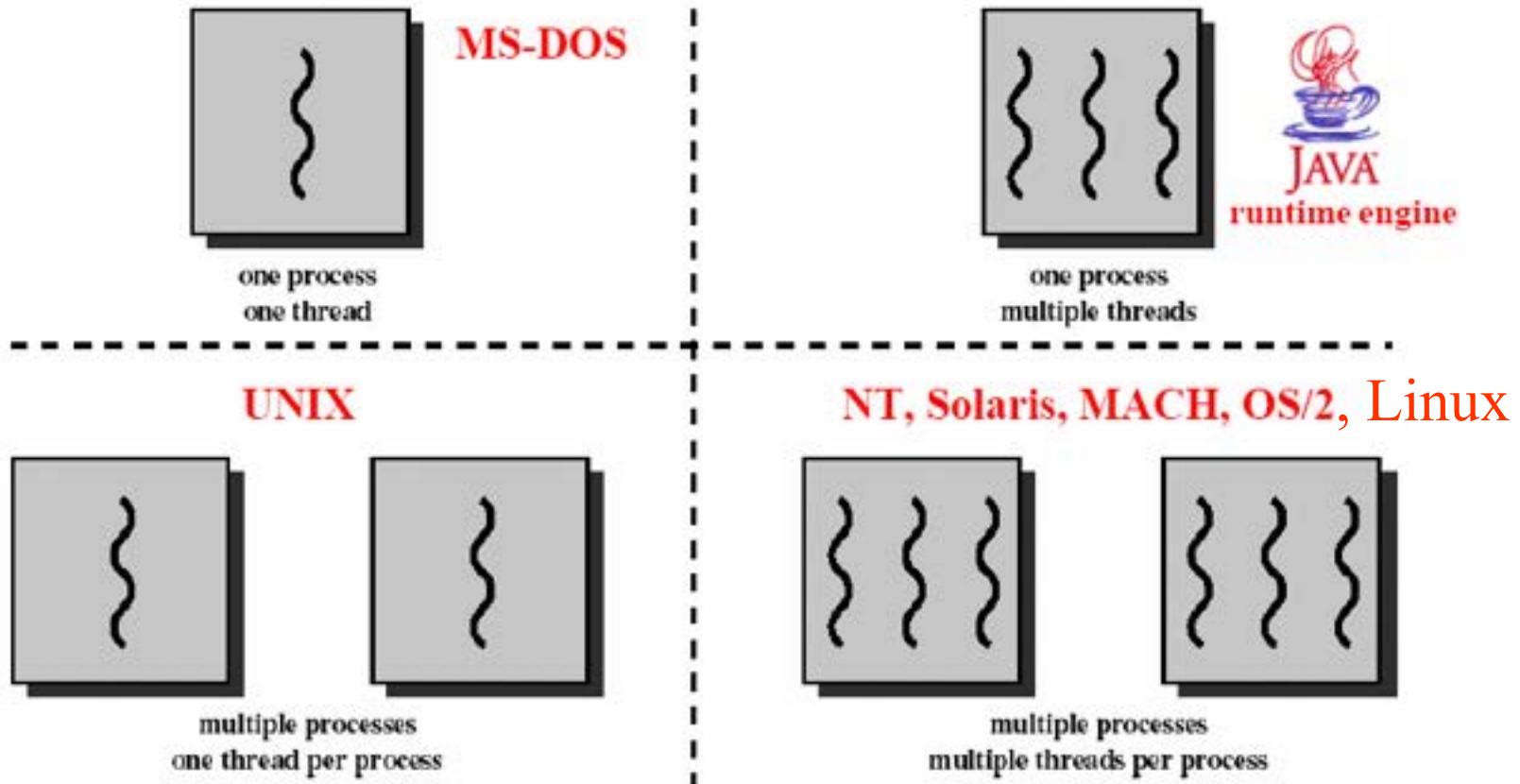
- **Condivisione di memoria:** a differenza dei processi (pesanti), un thread può condividere variabili con altri thread (appartenenti allo stesso task)
- **Minor costo di context switch:** PCB di thread non contiene alcuna informazione relativa a codice e dati globali:
-> il cambio di contesto fra thread dello stesso task ha un **costo notevolmente inferiore** al caso dei processi “pesanti”.

Svantaggio:

- **Minor protezione:** thread appartenenti allo stesso task possono modificare dati gestiti da altri thread

Soluzioni adottate

- **MS-DOS**: un solo processo utente ed un solo thread.
- **UNIX**: più processi utente ciascuno con un solo thread.
- Supporto run time di Java: un solo processo, più thread.
- **Linux, Windows, MacOSX, Solaris**: più processi utente ciascuno con più thread.



Realizzazione di thread

Molti SO offrono l'implementazione del concetto di thread (es. WinXP e successivi, Linux, Solaris, MacOSX,....)

Realizzazione

A livello kernel (WinXP, Linux, Solaris, MacOSX, versioni recenti java):

- SO gestisce direttamente i cambi di contesto
 - tra thread dello stesso task (trasferimento di registri)
 - tra task
- SO fornisce strumenti per la sincronizzazione per l'accesso di thread a variabili comuni

Realizzazione di thread

Realizzazione

A livello utente (es. Andrew - Carnegie Mellon, POSIX pthread, vecchie versioni MSWin, Java thread):

- il passaggio da un thread al successivo (nello stesso task) non richiede interruzioni al SO (**maggior rapidità**)
- SO vede processi pesanti: la sospensione di un thread causa la sospensione di tutti i thread del task

Soluzioni miste

- (es. Solaris)
- thread realizzati a entrambi i livelli

Interazione tra processi

I processi, pesanti o leggeri, possono interagire

Classificazione

- processi **indipendenti**: due processi P1 e P2 sono indipendenti se l'esecuzione di P1 non è influenzata da P2, e viceversa
- processi **interagenti**: P1 e P2 sono interagenti se l'esecuzione di P1 è influenzata dall'esecuzione di P2, e/o viceversa

Processi interagenti

Tipi di interazione:

- **Cooperazione**: interazione prevedibile e desiderata, insita nella logica del programma concorrente. I processi cooperanti collaborano per il raggiungimento di un fine comune
- **Competizione**: interazione prevedibile ma "non desiderata" tra processi che interagiscono per sincronizzarsi nell'accesso a risorse comuni
- **Interferenza**: interazione non prevista e non desiderata, potenzialmente deleteria tra processi

Processi interagenti

Supporto all'interazione

L'interazione può avvenire mediante:

- **memoria condivisa** (modello ad **ambiente globale**): SO consente ai processi (thread) di condividere variabili; l'interazione avviene tramite l'accesso a variabili condivise. Il SO prevede dei meccanismi per imporre dei vincoli di sincronizzazione nell'accesso alle variabili condivise
- **scambio di messaggi** (modello ad **ambiente locale**): i processi non condividono variabili e interagiscono mediante meccanismi di trasmissione/ricezione di messaggi. Il SO prevede dei meccanismi a supporto dello scambio di messaggi

Processi interagenti

Aspetti

- concorrenza -> velocità
- suddivisione dei compiti tra processi -> modularità
- condivisione di informazioni
 - assenza di replicazione: ogni processo accede alle stesse istanze di dati
 - necessità di sincronizzare i processi nell'accesso a dati condivisi

Competizione: mutua esclusione

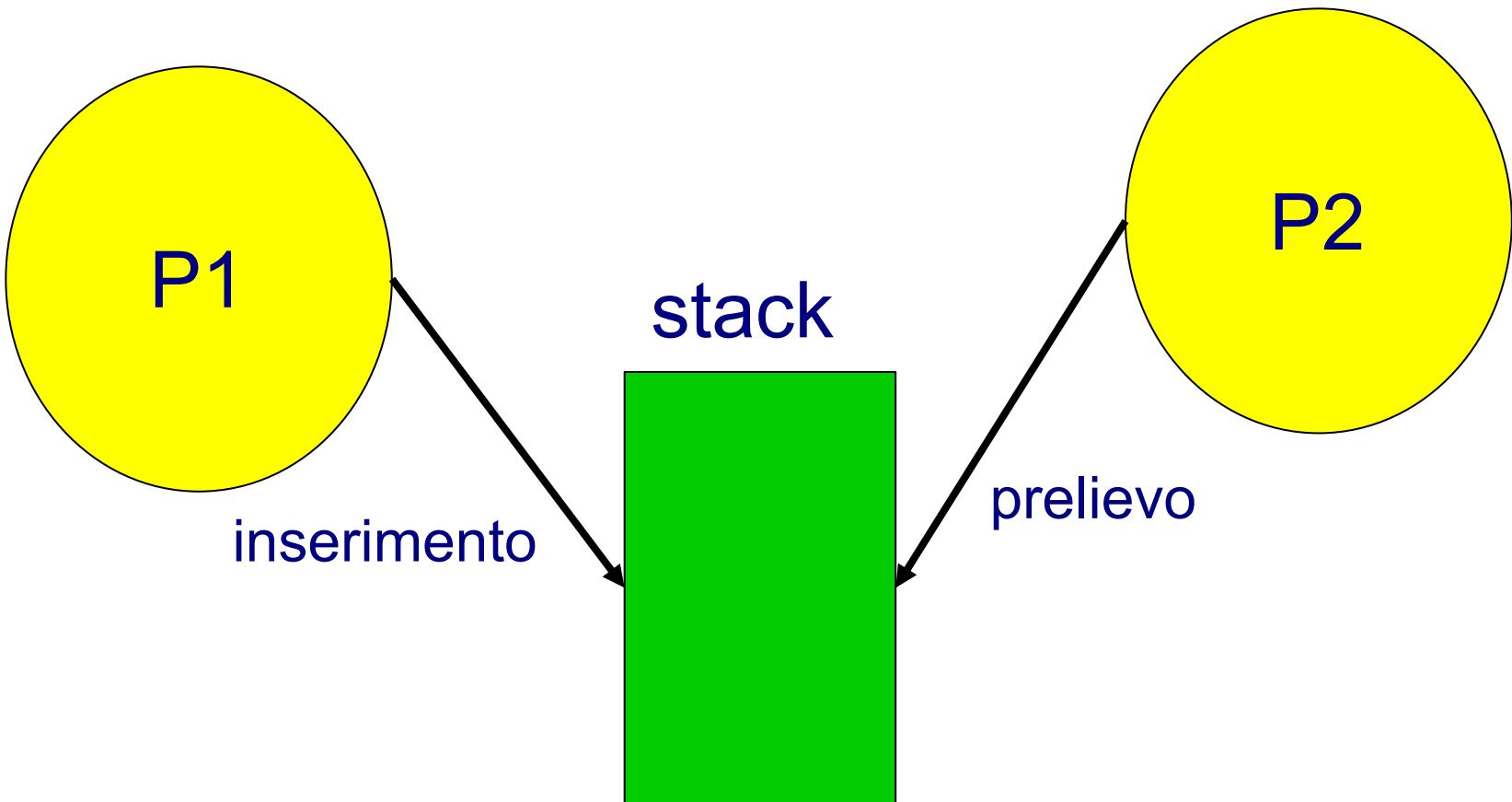
Esempio

- Due thread P1 e P2 hanno accesso ad una struttura condivisa organizzata a pila rispettivamente per inserire e prelevare dati.
- La struttura dati è rappresentata da un vettore stack i cui elementi costituiscono i singoli dati e da una variabile top che indica la posizione dell'ultimo elemento contenuto nella pila.
- I thread utilizzano le operazioni Inserimento e Prelievo per depositare e prelevare i dati dalla pila.

```
typedef ... item;
item stack[N];
int top=-1;

void Inserimento(item y)
{
    top++;
    stack[top]=y;
}

item Prelievo()
{
    item x;
    x= stack[top];
    top--;
    return x;
}
```



HP: P1 e P2 eseguono concorrentemente operazioni (inserimento e prelievo rispettivamente) sullo stack condiviso.

- L'esecuzione contemporanea di queste operazioni da parte dei processi può portare ad un uso scorretto della risorsa.

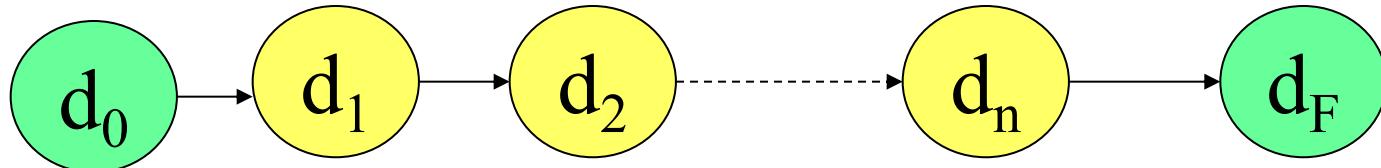
- Consideriamo questa sequenza di esecuzione:

T0:	top++;	(P1)
T1:	x=stack [top] ;	(P2)
T2:	top--;	(P2)
T3:	stack [top]=y;	(P1)

- Viene assegnato a x un valore non definito e l'ultimo valore valido contenuto nella pila viene cancellato dal nuovo valore di y.
- Potremmo individuare situazioni analoghe, nel caso di esecuzione contemporanea di una qualunque delle due operazioni da parte dei due processi.
- Il problema sarebbe risolto se le due operazioni di Inserimento e Prelievo fossero eseguite sempre in **mutua esclusione** (istruzioni **indivisibili**): finchè non è terminata l'operazione corrente, nessun'altra operazione può essere eseguita sullo stack.

Istruzioni Indivisibili

- Data un'istruzione $I(d)$, che opera su un dato d , essa e` **indivisibile** (o **atomica**) , se, durante la sua esecuzione da parte di un processo P, il dato d non e` accessibile ad altri processi.
- $I(d)$, a partire da un valore iniziale d_0 , puo`operare delle trasformazioni sullo stato di d , fino a giungere allo stato finale d_F ;
→ poiche` $I(d)$ e` indivisibile, gli stati intermedi non possono essere rilevati da altri processi concorrenti.



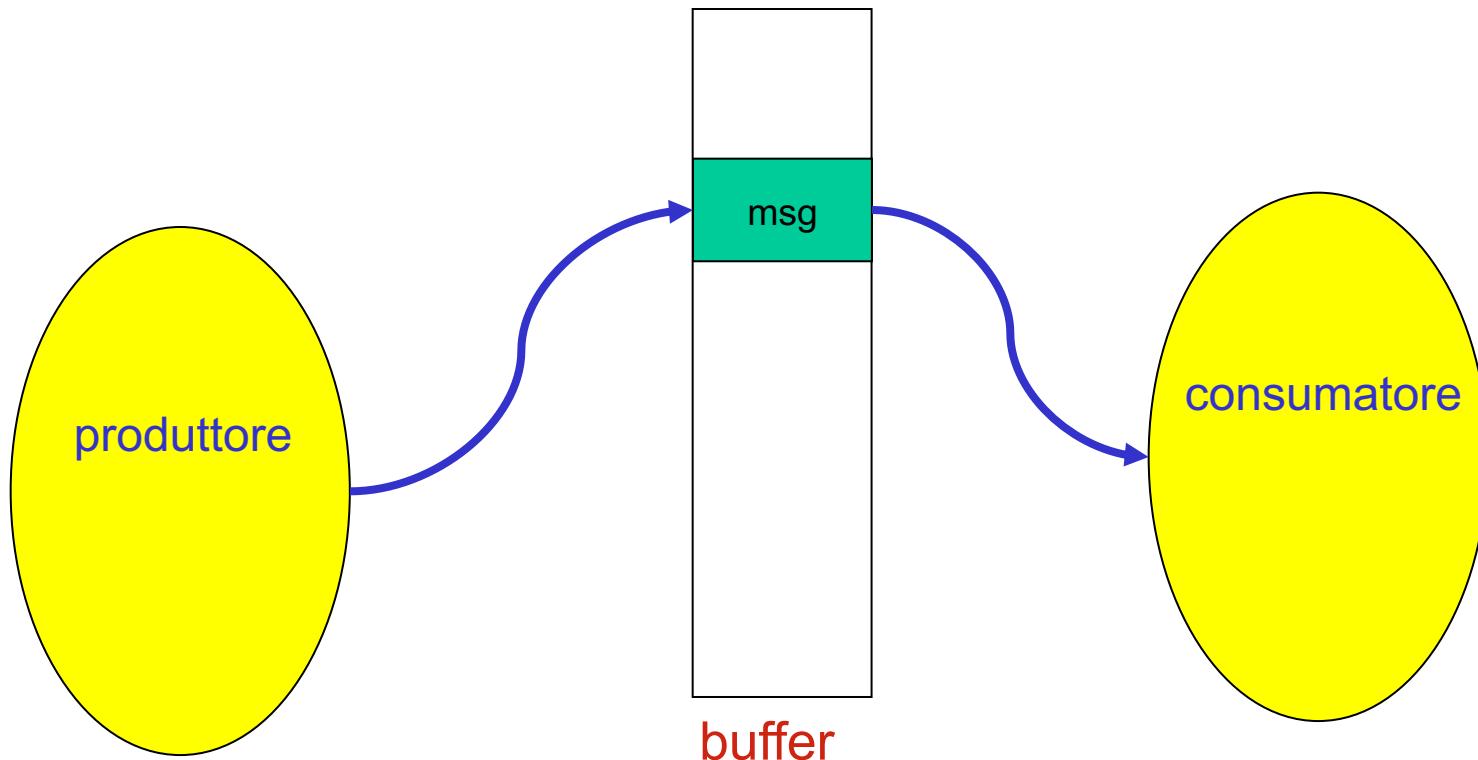
Processi cooperanti

Esempio: produttore & consumatore

Due thread accedono a un buffer condiviso di dimensione limitata:

- un processo svolge il ruolo di **produttore** di informazioni che verranno prelevate dall'altro processo (**consumatore**)
- buffer rappresenta un deposito di informazioni condiviso

Produttore & consumatore



Produttore & consumatore

Necessità di sincronizzare i processi:

- quando il **buffer è vuoto** (il consumatore NON può prelevare messaggi)
- quando il **buffer è pieno** (il produttore NON può depositare messaggi)

Produttore & consumatore

Processo produttore:

```
....  
shared msg Buff [DIM];  
main()  
{ msg M;  
  do  
  { produco(&M);  
    inserisco(M, Buff);  
  } while(!fine);  
}
```

Processo consumatore:

```
....  
shared msg Buff [DIM];  
main()  
{ msg M;  
  do  
  { prelievo(&M, Buff);  
    consumo(M);  
  } while(!fine);  
}
```

Problema: che cosa succede se

- il buffer è pieno?
- il buffer è vuoto?

Produttore & consumatore

DIM=1

Necessità di sincronizzare i processi

Aggiungiamo due variabili logiche condivise:

- **buff_vuoto**: se uguale a true, indica che il buffer non contiene messaggi (viene settata dalla funzione di prelievo quando l'unico messaggio presente nel buffer viene estratto)
- **buff_pieno**: se uguale a true, indica che il buffer non può accogliere nuovi messaggi, perché pieno (viene settata dalla funzione di inserimento quando l'ultima locazione libera del buffer viene riempita)

Produttore & consumatore: buffer dim=1

Processo produttore:

```
....  
shared int buff_pieno=0;  
shared int buff_vuoto=1;  
shared msg Buff [DIM];  
main()  
{ msg M;  
  do  
  { produco(&M);  
    while (buffer_pieno);  
    buffer_pieno=1;  
    inserisco(M, Buff);  
    buffer_vuoto=0;  
  } while(!fine);  
}
```

Processo consumatore:

```
....  
shared int buff_pieno=0;  
shared int buff_vuoto=1;  
shared msg Buff [DIM];  
main()  
{ msg M;  
  do  
  { while (buffer_vuoto);  
    buffer_vuoto=1;  
    prelievo(M, Buff);  
    buffer_pieno=0;  
    consumo(&M);  
  } while(!fine);  
}
```

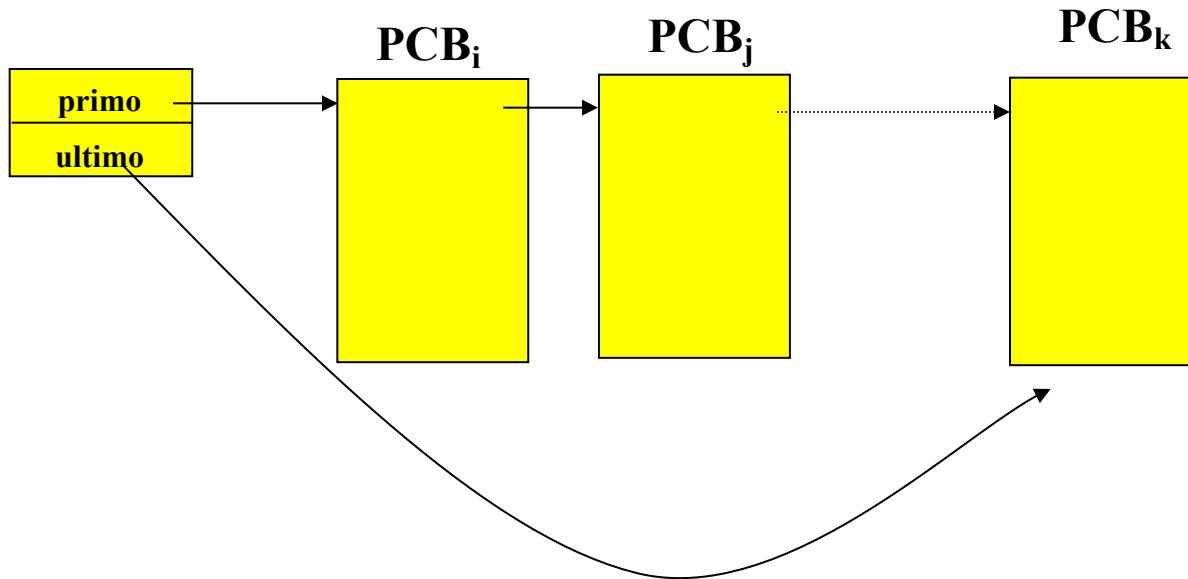
Scheduling della CPU

Scheduling della CPU

Obiettivo della multiprogrammazione:
massimizzazione dell'utilizzo CPU

- **Scheduling della CPU:**
commuta l'uso della CPU tra i vari processi
- **Scheduler della CPU (a breve termine):** è quella parte del SO che seleziona dalla coda dei processi pronti il prossimo processo al quale assegnare l'uso della CPU

Coda dei processi pronti (ready queue):



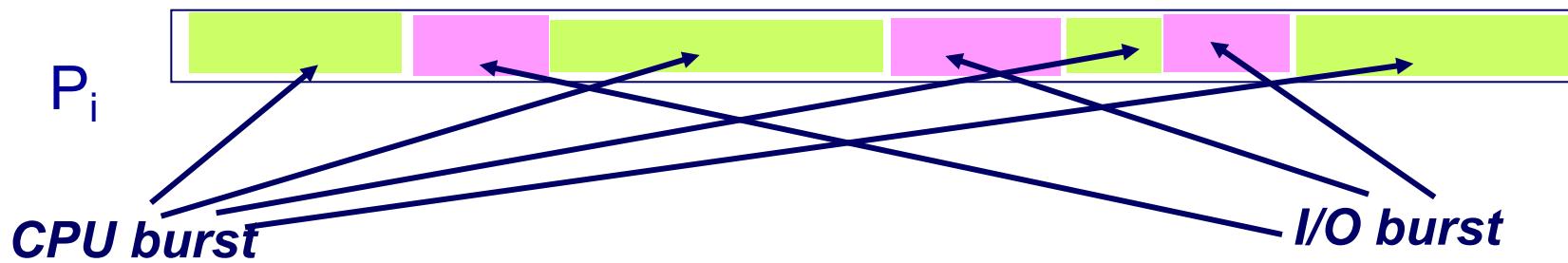
contiene i descrittori (process control block, PCB) dei processi pronti

La strategia di gestione della ready queue è realizzata mediante **politiche** (algoritmi) di scheduling

Terminologia: CPU burst & I/O burst

Ogni processo alterna: (burst = raffica)

- **CPU burst:** fasi in cui viene impiegata soltanto la CPU senza interruzioni dovute a operazioni di I/O
- **I/O burst:** fasi in cui il processo effettua I/O da/verso un dispositivo



- Quando un processo è in I/O burst, la CPU non viene utilizzata: in un sistema multiprogrammato, lo short-term scheduler assegna la CPU a un nuovo processo

Terminologia: processi I/O bound & CPU bound

A seconda delle caratteristiche dei programmi eseguiti dai processi, è possibile classificare i processi in:

- **I/O bound**: prevalenza di attività di I/O
 - Molti CPU burst di breve durata, intervallati da I/O burst di lunga durata
- **CPU bound**: prevalenza di attività di computazione
 - CPU burst di lunga durata, intervallati da pochi I/O burst di breve durata

Terminologia: pre-emption

Gli algoritmi di scheduling si possono classificare in due categorie:

- senza prelazione (**non pre-emptive**): CPU rimane allocata al processo running finché esso non si sospende volontariamente o non termina
- con prelazione (**pre-emptive**): processo running può essere prelazionato, cioè SO può sottrargli CPU per assegnarla ad un nuovo processo

➤ I sistemi a **divisione di tempo** hanno sempre uno scheduling **pre-emptive**

Politiche & meccanismi

Lo **scheduler** decide a quale processo assegnare la CPU; a seguito della decisione, viene attuato il cambio di contesto (context-switch)

Dispatcher : è la parte di SO che realizza il cambio di contesto

Scheduler = POLITICHE

Dispatcher = MECCANISMI

Criteri di scheduling

Per analizzare e confrontare i diversi algoritmi di scheduling, vengono considerati alcuni indicatori di performance:

- **Utilizzo della CPU**: percentuale media di utilizzo CPU nell' unità di tempo
- **Throughput** (del sistema): numero di processi completati nell'unità di tempo
- **Tempo di Attesa** (di un processo): tempo totale trascorso nella ready queue
- **Turnaround** (di un processo): tempo tra la sottomissione del job e il suo completamento
- **Tempo di Risposta** (di un processo): intervallo di tempo tra la sottomissione e l'inizio della prima risposta (a differenza del turnaround, non dipende dalla velocità dei dispositivi di I/O)

Criteri di scheduling

In generale:

- devono essere **massimizzati**
 - Utilizzo della CPU
 - Throughput
- invece, devono essere **minimizzati**
 - Turnaround (sistemi *batch*)
 - Tempo di Attesa
 - Tempo di Risposta (sistemi *interattivi*)

Criteri di scheduling

Non è possibile soddisfare tutti i criteri contemporaneamente

A seconda del tipo di SO, gli algoritmi di scheduling possono avere diversi obiettivi.

Ad esempio:

- nei sistemi **batch**:
 - massimizzare throughput e minimizzare turnaround
- nei sistemi **interattivi**:
 - minimizzare il tempo medio di risposta dei processi
 - minimizzare il tempo di attesa

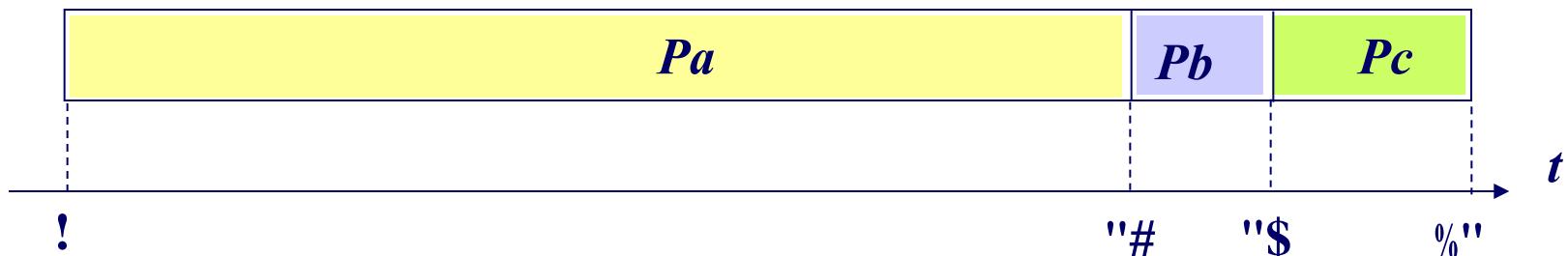
Alcuni algoritmi di scheduling

Algoritmo di scheduling FCFS

First-Come-First-Served: la coda dei processi pronti viene gestita in modo FIFO

- i processi sono schedulati secondo l'ordine di arrivo nella coda
- algoritmo **non pre-emptive**

Esempio: tre processi [Pa, Pb, Pc] (diagramma di Gantt)



$$T_{avg} = (0 + 32 + 36)/3 = 22,7$$

Problemi dell'algoritmo FCFS

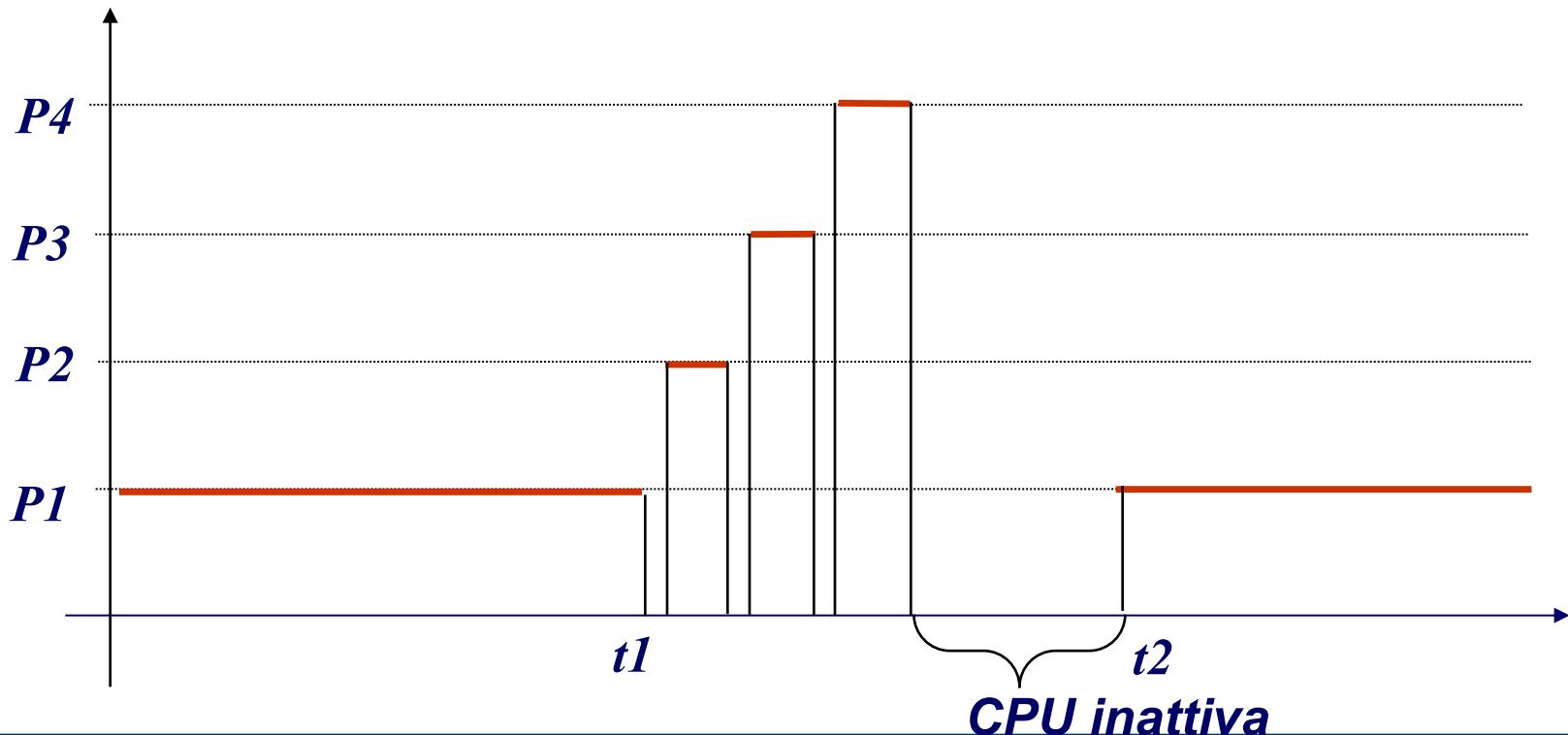
Non è possibile influire sull'ordine dei processi:

- nel caso di processi in attesa dietro a processi con lunghi CPU burst (processi CPU bound), il tempo di attesa è alto
- Possibilità di effetto convoglio
se molti processi I/O bound seguono un processo CPU bound: scarso grado di utilizzo della CPU

Algoritmo di scheduling FCFS: effetto convoglio

Esempio: [P1, P2, P3, P4]

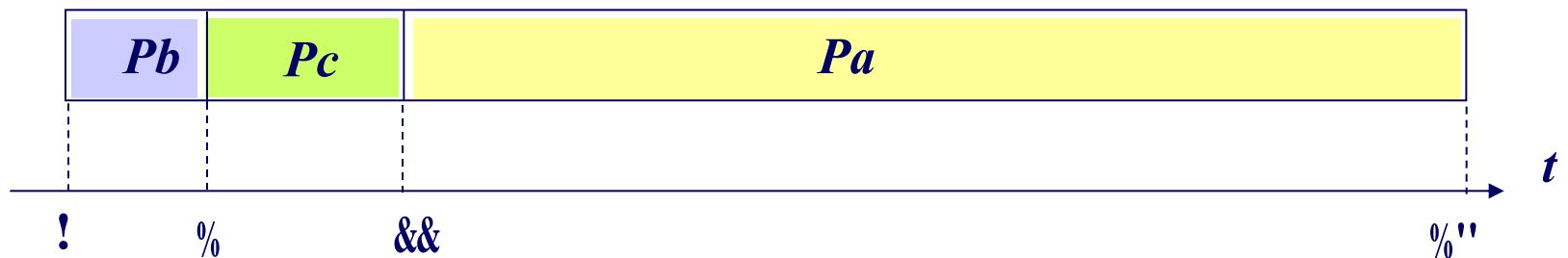
- P1 è CPU bound; P2, P3, P4 sono I/O bound
- P1 effettua I/O nell'intervallo $[t1, t2]$



Algoritmo di scheduling SJF (Shortest Job First)

Per risolvere i problemi dell'algoritmo FCFS:

- per ogni processo nella ready queue, viene stimata la lunghezza del prossimo CPU-burst
- viene schedulato il processo con il CPU burst più corto (Shortest Job First)



$$T_{\text{attesa medio}} = (0 + 4 + 11)/3 = 5$$

- si può dimostrare che questo algoritmo ottimizza il tempo di attesa

Algoritmo di scheduling SJF (Shortest Job First)

SJF può essere:

- non pre-emptive
- pre-emptive: (Shortest Remaining Time First, SRTF) se nella coda arriva un processo (Q) con CPU burst minore del CPU burst rimasto al processo running (P) ➔ pre-emption

Problema

- è difficile stimare la lunghezza del prossimo CPU burst di un processo (di solito, uso del passato per predire il futuro)

Stimare la lunghezza di CPU burst

Approccio approssimato: stimare probabilisticamente la lunghezza del CPU Burst in funzione dai precedenti CPU burst di quel processo

Tecnica usata frequentemente: exponential averaging

t_n = actual length of n^{th} CPU burst

τ_{n+1} = predicted value for the next CPU burst

$\alpha, 0 \leq \alpha \leq 1$

$$\boxed{\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n}$$

SJF con exponential averaging

Sviluppando l' espressione:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots + (1 - \alpha)^{j-1} \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$

→ ogni termine ha meno peso del termine precedente

- $\alpha = 0$

$$\tau_{n+1} = \tau_n$$

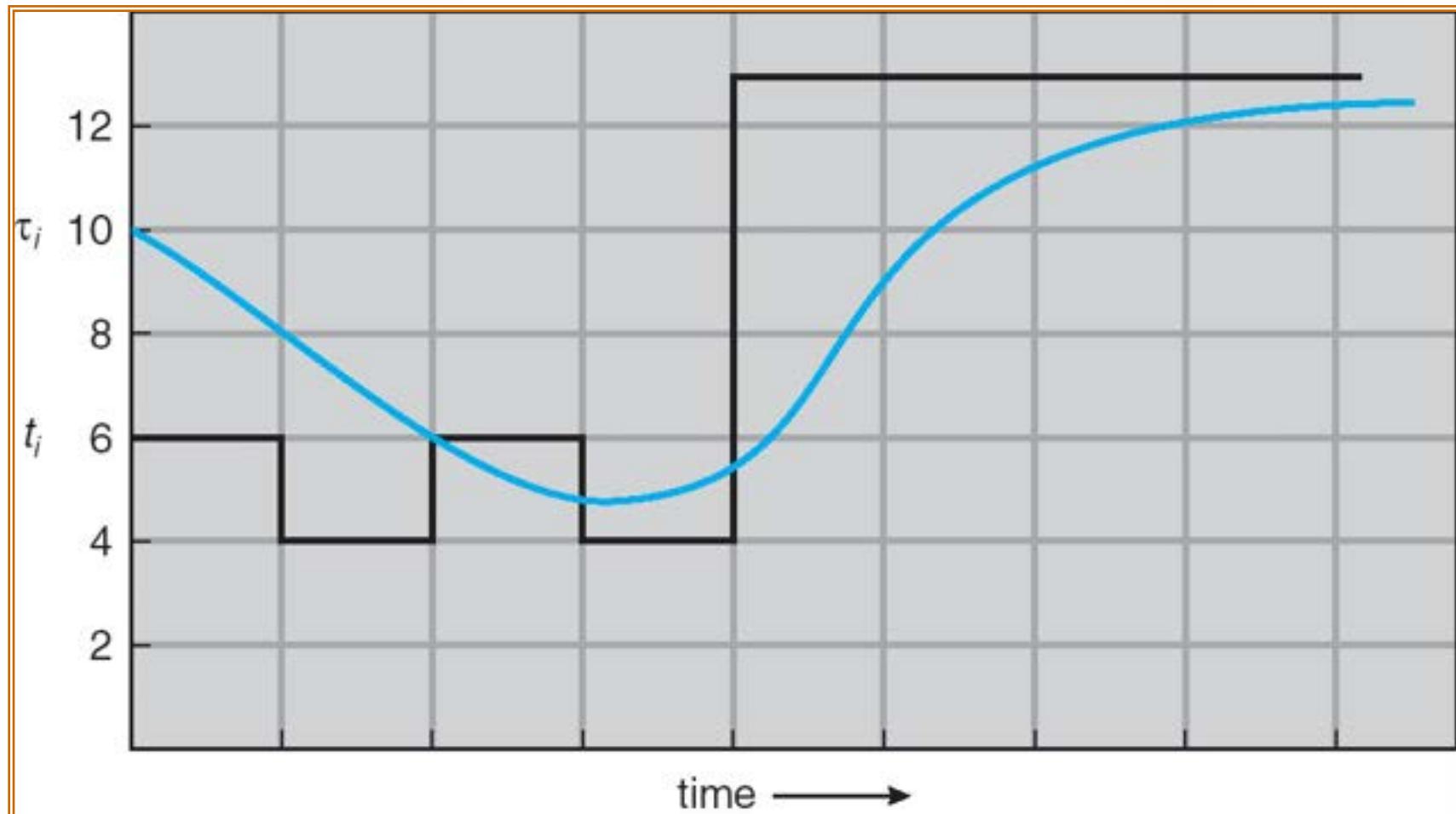
ovvero la storia recente non conta (il valore di t_n non influisce)

- $\alpha = 1$

$$\tau_{n+1} = \alpha t_n$$

ovvero conta solo l' ultimo valore reale

Stimare la lunghezza di CPU burst ($\alpha=1/2$)



CPU burst (t_i)

"guess" (τ_i) 10

6

8

4

6

6

6

4

5

13

9

13

11

13

12

...

...

Scheduling con priorità

Ad ogni processo viene assegnata una priorità:

- lo scheduler seleziona il processo pronto con priorità massima
- processi con uguale priorità vengono trattati in modo FCFS

Priorità possono essere definite:

- internamente: SO attribuisce ad ogni processo una priorità in base a politiche interne
- esternamente: criteri esterni al SO (es: `nice` in UNIX)

Le priorità possono essere costanti o variare dinamicamente.

Scheduling con priorità

Problema: starvation dei processi

Starvation: si verifica quando uno o più processi di priorità bassa vengono lasciati indefinitamente nella coda dei processi pronti, perché vi è sempre almeno un processo pronto di priorità più alta.

Soluzione: modifica dinamica della priorità dei processi.

Ad esempio:

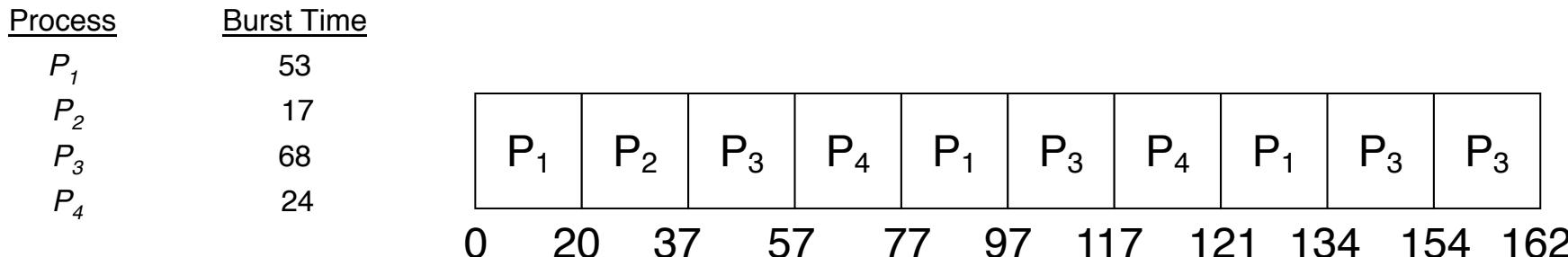
- la priorità decresce al crescere del tempo di CPU già utilizzato (feedback negativo o aging)
- la priorità cresce dinamicamente con il tempo di attesa del processo (feedback positivo o promotion)

Algoritmo di scheduling Round Robin

Algoritmo preemptive tipicamente usato in sistemi time sharing:

- Ready queue gestita come una coda FIFO circolare (FCFS)
- ad ogni processo viene allocata la CPU per un intervallo di tempo costante Δt (time slice o, quanto di tempo)
 - il processo usa la CPU per Δt (oppure si blocca prima)
 - allo scadere del quanto di tempo: revoca della CPU e re-inserimento in coda

Esempio: $\Delta t = 20\text{ms}$



- RR può essere visto come un'estensione di FCFS con pre-emption periodica

Round Robin

- Obiettivo principale è la minimizzazione del tempo di risposta (adeguato per sistemi interattivi)
- Tutti i processi sono trattati allo stesso modo (assenza di starvation)

Problemi:

- dimensionamento del quanto di tempo
 - Δt piccolo (ma non troppo: $\Delta t \gg T_{\text{context switch}}$) : tempi di risposta ridotti, ma alta frequenza di context switch => overhead eccessivo
 - Δt grande: overhead di context switch ridotto, ma tempi di risposta più alti
- trattamento equo di tutti i processi
 - possibilità di degrado delle prestazioni del SO

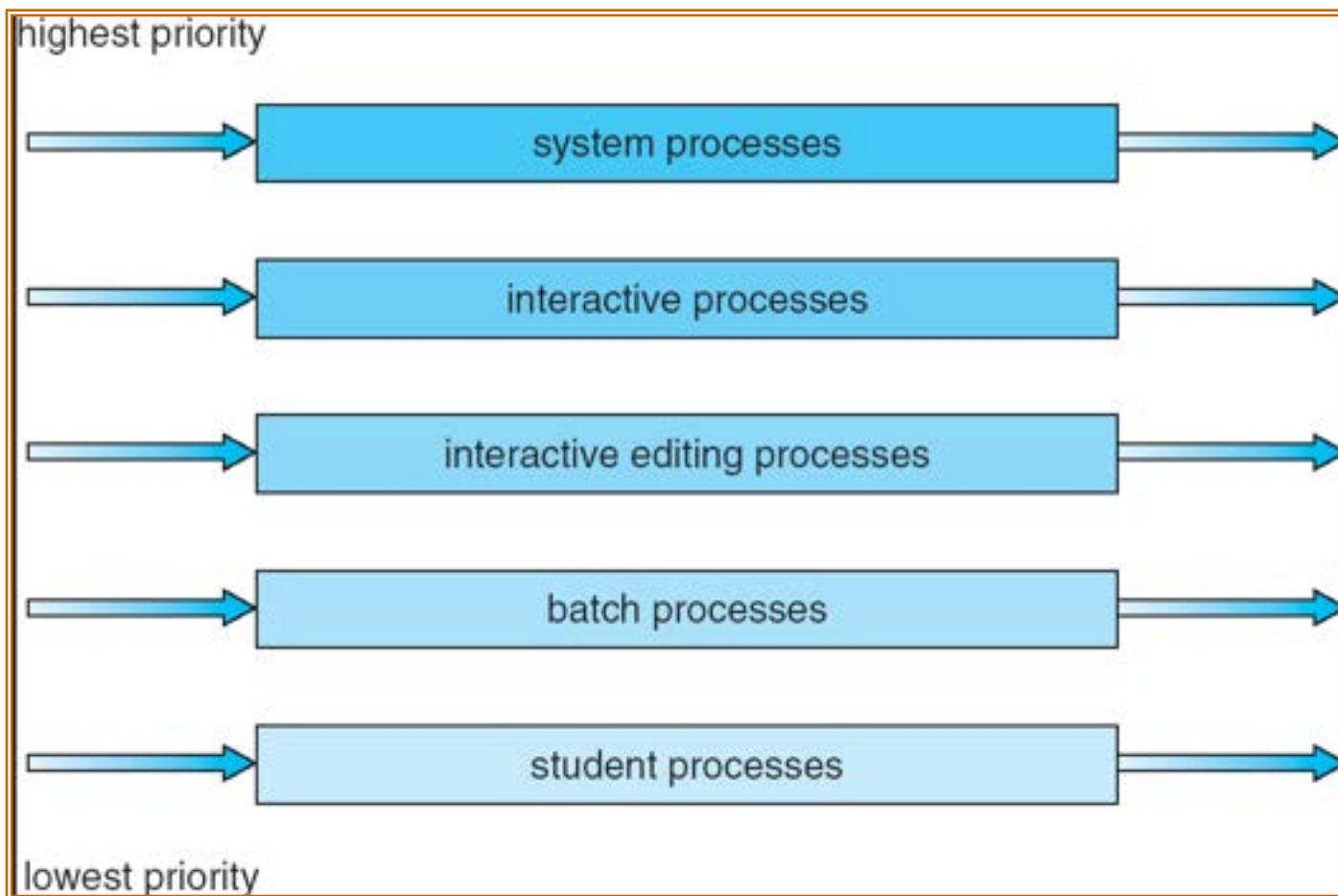
Approcci misti

Nei SO reali, spesso si combinano diversi algoritmi di scheduling

Esempio: Multiple Level Feedback Queues

- ❑ più code, ognuna associata a un tipo di job diverso (batch, interactive, CPU-bound, ...)
- ❑ ogni coda ha una diversa priorità: scheduling delle code con priorità
- ❑ ogni coda viene gestita con un algoritmo di scheduling distinto, eventualmente diverso (es. FCFS o Round Robin)
- ❑ i processi possono muoversi da una coda all'altra, in base alla loro storia:
 - passaggio da priorità bassa ad alta: processi in attesa da molto tempo (feedback positivo)
 - passaggio da priorità alta a bassa: processi che hanno già utilizzato molto tempo di CPU (feedback negativo)

Multi Level Feedback Queue



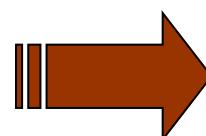
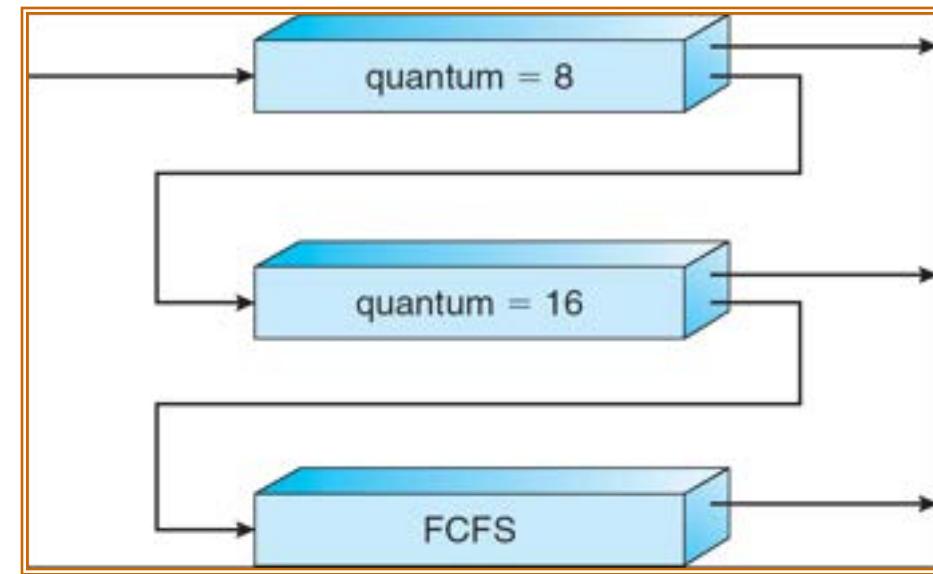
Esempio di Multi Level Feedback Queue

3 code

- Q_0 – RR con time quantum=8ms
- Q_1 – RR con time quantum=16ms
- Q_2 – FCFS

Scheduling

- Un processo nuovo entra in Q_0 ; quando acquisisce la CPU ha 8ms per utilizzarla; se non termina nel quanto di tempo viene spostato in Q_1
- In Q_1 il processo è servito ancora RR e riceve 16ms di CPU; se non termina nel quanto di tempo, viene spostato in Q_2



*Priorità elevata a
processi con breve
uso CPU*

Scheduling in UNIX (BSD 4.3)

Obiettivo: privilegiare i processi interattivi

Scheduling MLFQ:

- più livelli di priorità (circa 160): la priorità è rappresentata da un intero; più grande è il suo valore, più bassa è la priorità
- Viene definito un valore di riferimento **pzero**:
 - ✓ Priorità $\geq pzero$: processi di utente ordinari
 - ✓ Priorità $< pzero$: processi di sistema (ad es. esecuzione di system call)
- Ad ogni livello è associata una coda, gestita Round Robin (quanto di tempo: 100 ms)

Scheduling in UNIX

- Aggiornamento dinamico delle priorità: ad ogni secondo viene ricalcolata la priorità di ogni processo
- La priorità di un processo decresce al crescere del tempo di CPU già utilizzato
 - feedback negativo
 - di solito, processi interattivi usano poco la CPU: in questo modo vengono favoriti
- L'utente può influire sulla priorità: comando **nice** (ovviamente soltanto per decrescere la priorità)

I Processi nel SO UNIX

Processi UNIX

UNIX è un sistema operativo multiprogrammato a divisione di tempo:

unità di computazione è il processo

**Caratteristiche del processo UNIX:
processo pesante con codice rientrante:**

- » dati non condivisi
- » codice condivisibile con altri processi

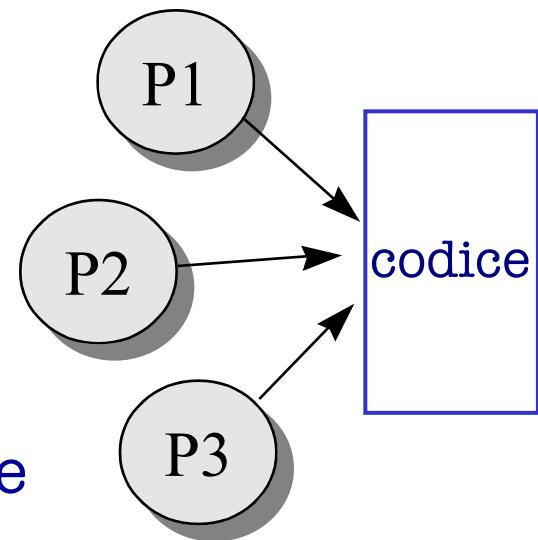
Modello di processo in UNIX

Ogni processo ha un proprio spazio di indirizzamento **locale e non condiviso**:

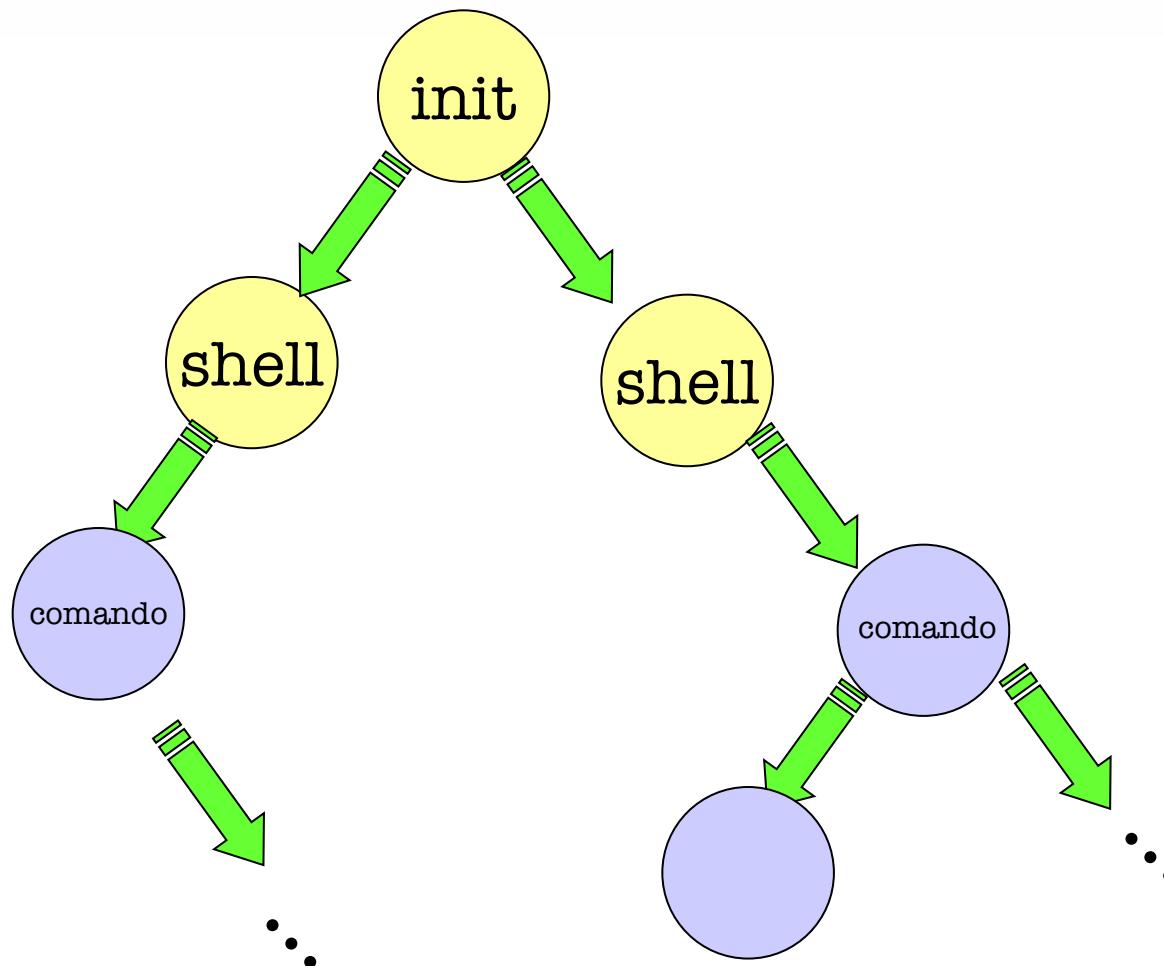
**Modello ad Ambiente Locale
(o a scambio di messaggi)**

Eccezione:

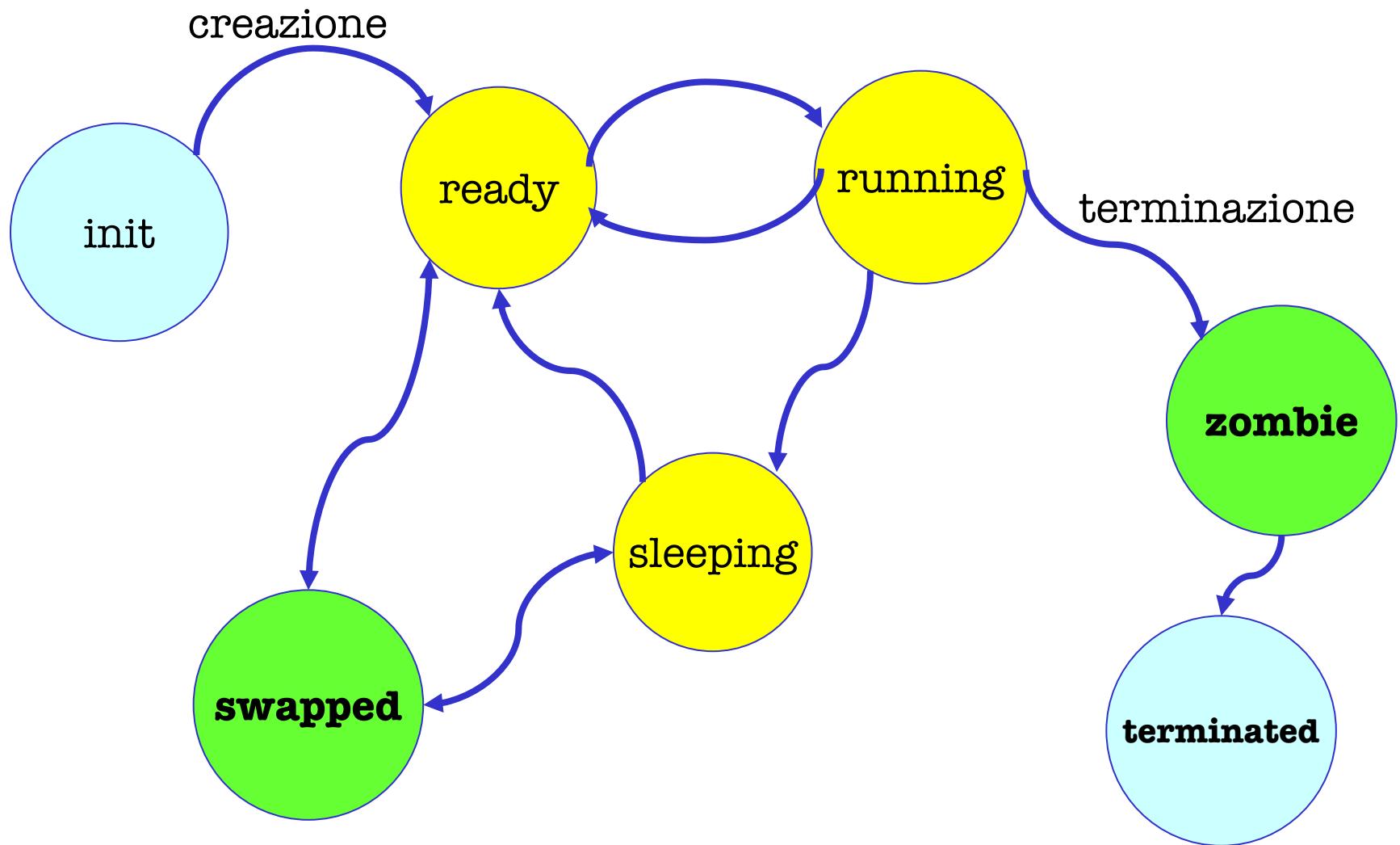
- il **codice** può essere **condiviso** (codice rientrante)



Gerarchie di processi UNIX



Stati di un processo UNIX



Stati di un processo UNIX

Come nel caso generale:

- ❑ **Init**: caricamento in memoria del processo e inizializzazione delle strutture dati del SO
- ❑ **Ready**: processo pronto
- ❑ **Running**: il processo **usa la CPU**
- ❑ **Sleeping** : il processo è **sospeso in attesa di un evento (v. waiting)**
- ❑ **Terminated**: **eliminazione** del processo dalla memoria e dal SO

In aggiunta:

- ❑ **Zombie**: il processo è terminato, ma è **in attesa che il padre ne rilevi lo stato di terminazione**
- ❑ **Swapped**: il processo (o parte di esso) è **temporaneamente trasferito in memoria secondaria**

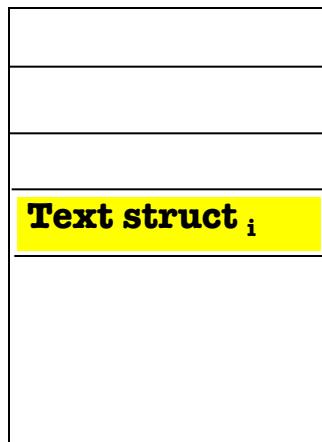
Processi swapped

Lo scheduler a medio termine (swapper) gestisce i trasferimenti dei processi

- **da memoria centrale a secondaria (dispositivo di swap): swap out**
 - ✓ si applica preferibilmente ai **processi bloccati (sleeping)**, prendendo in considerazione tempo di attesa, di permanenza in memoria e dimensione del processo (preferibilmente i **processi più lunghi**)
- **da memoria secondaria a centrale: swap in**
 - ✓ si applica preferibilmente ai **processi più corti**

Rappresentazione dei processi UNIX

Il codice dei processi è rientrante: più processi possono condividere lo stesso codice (text):



Text table:

1 elemento per ogni programma (codice) utilizzato

- ✓ codice e dati sono separati (modello a codice puro)
- ✓ SO gestisce una **struttura dati globale** in cui sono contenuti i **puntatori ai codici utilizzati**, eventualmente condivisi) dai processi: **text table**
- ✓ L'elemento della text table si chiama **text structure** e contiene:
 - » **puntatore al codice** (se il processo è swapped, riferimento a memoria secondaria)
 - » **numero** dei processi che lo condividono

Rappresentazione dei processi UNIX

Process control block: il descrittore del processo in UNIX è rappresentato da **2 strutture dati distinte**:

- **Process structure:** informazioni necessarie **al sistema per la gestione del processo** (a prescindere dallo stato in cui si trova il processo)
- **User structure:** informazioni necessarie solo se il processo è **residente in memoria centrale**

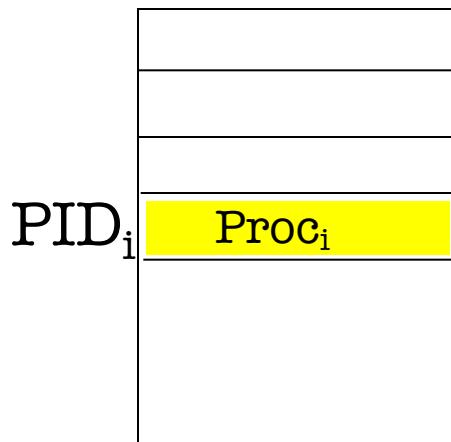
Process structure

Process structure contiene, tra le altre, le seguenti informazioni:

- Un valore intero non negativo che rappresenta l'identificatore unico del processo (**Process IDentifier, PID**)
- Lo **stato** del processo
- **puntatori alle aree dati e stack** associati al processo
- riferimento indiretto al **codice**, ovvero il riferimento all'elemento della text table (text structure) associato al codice del processo
- informazioni di **scheduling** (es: priorità, tempo di CPU, ...)
- riferimento al **processo padre** ovvero il PID del padre;
- informazioni relative alla **gestione di segnali** (es. segnali inviati ma non ancora gestiti, v. segnali..)
- Puntatore al processo successivo nella **coda** di processi (ad esempio, ready queue) alla quale il processo eventualmente appartiene
- puntatore alla **User Structure**

Rappresentazione dei processi UNIX

Tutte le Process structure sono organizzate in un vettore gestito dal sistema operativo: **Process table**



Process table: 1 elemento per ogni processo

User structure

Contiene le informazioni necessarie al SO per la gestione del processo, **quando è residente in memoria** (ovvero non è swapped):

- copia dei **registri** di CPU (Program Counter, ecc.)
- informazioni sulle risorse allocate (**file aperti**)
- informazioni sulla gestione di **segnali** (puntatori a handler, ecc., v. segnali..)
- **ambiente** del processo: direttorio corrente, utente, gruppo, argc/argv, path, ...

Immagine di un processo UNIX

L'immagine di un processo è l'insieme delle aree di memoria e strutture dati associate al processo.

- **Protezione:** Non tutta l'immagine è accessibile in modo user:
 - parte di **kernel**
 - parte di **utente**
- **Presenza in memoria centrale:** Ogni processo può essere soggetto a swapping: in questo caso non tutta l'immagine può essere trasferita in memoria
 - parte **swappable**
 - parte **residente** o **non swappable**

Immagine di un processo UNIX

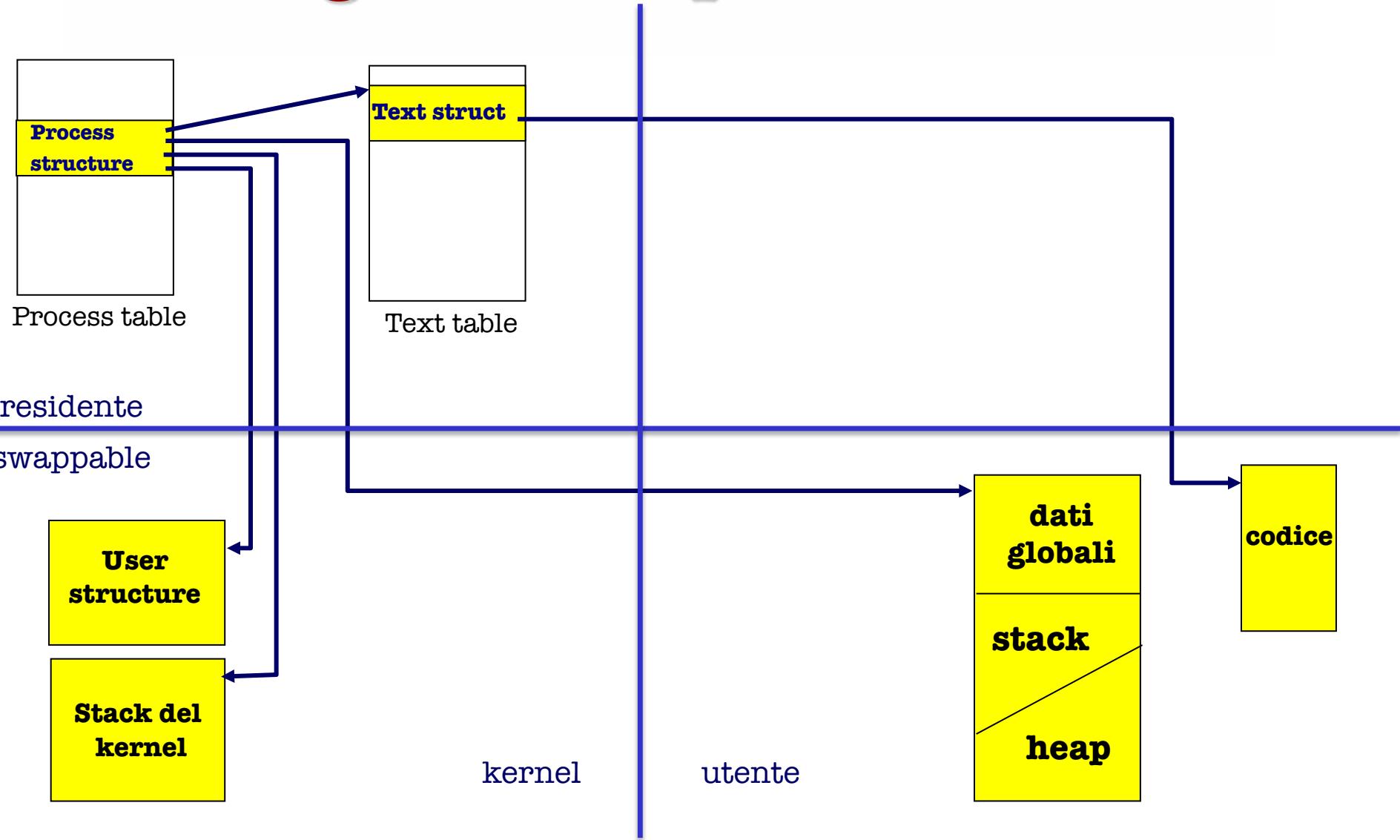


Immagine di un processo UNIX

Componenti

- **process structure**: è l'elemento della process table associato al processo (kernel, residente)
- **user structure**: struttura dati contenente i dati necessari al kernel per la gestione del processo quando è residente (kernel, swappable)
- **text**: elemento della text table associato al codice del processo (kernel, residente)
- area di **codice**: contiene il codice del programma eseguito dal processo (user, swappable)
- area **dati globali di utente**: contiene le **variabili globali** del programma eseguito dal processo (user, swappable)
- **stack, heap** di utente: **aree dinamiche** associate al programma eseguito (user, swappable)
- **stack del kernel**: stack di sistema associato al processo per le chiamate a **system call** (kernel, swappable)

PCB = process structure + user structure

- **Process structure:** risiede sempre in memoria centrale e mantiene le informazioni necessarie per la gestione del processo in ogni stato (anche se il processo è **swapped**)
- **User structure:** il suo contenuto è necessario **solo in caso di esecuzione** del processo (stato **running**): se il processo è soggetto a swapping, anche **la user structure può essere trasferita in memoria secondaria**

La **Process structure** contiene il riferimento alla **User structure** (in memoria centrale, o secondaria se swapped)

System call per la gestione di processi

Chiamate di sistema per:

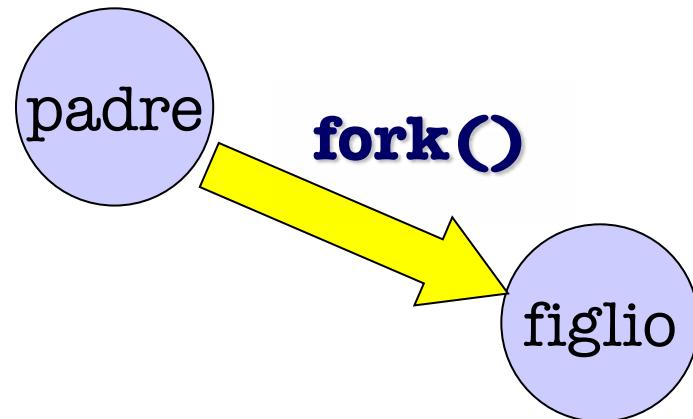
- **creazione di processi:** `fork()`
- **terminazione:** `exit()`
- **sospensione** in attesa della **terminazione di figli:** `wait()`
- **sostituzione di codice e dati:** `exec...()`

N.B. System call di UNIX sono attivabili attraverso chiamate a funzioni di librerie C standard: `fork()`, `exec()`, ... sono quindi funzioni di libreria che chiamano le system call corrispondenti

Creazione di processi: fork()

La funzione **fork()** consente a un processo di **generare un processo figlio**:

- padre e figlio **condividono lo STESSO codice**
- il figlio **EREDITA** una **copia dei dati (di utente e di kernel)** del padre



fork()

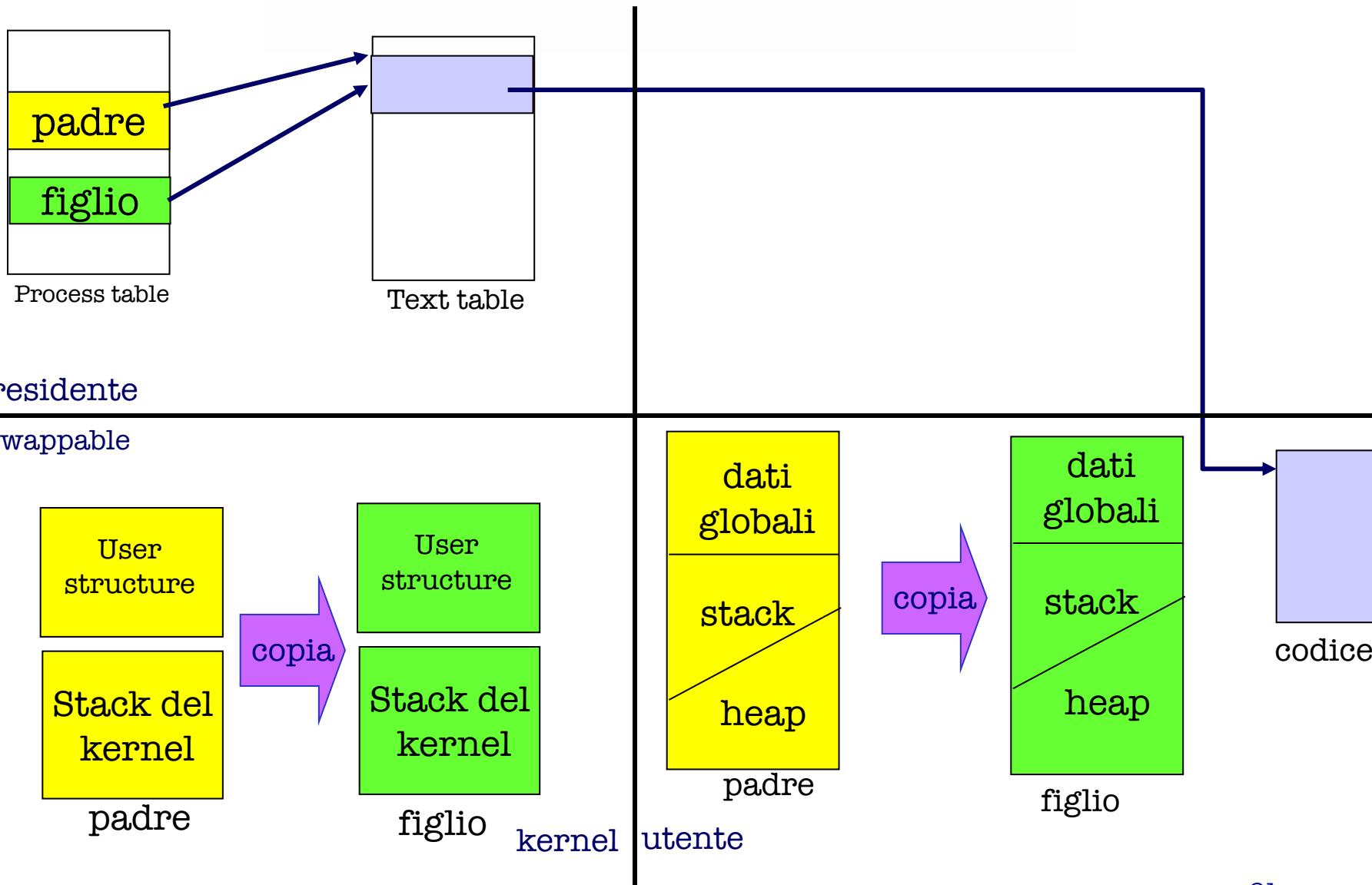
```
int fork(void);
```

- **fork()** non richiede parametri
- restituisce un **intero** che:
 - per il processo creato (**figlio**) vale **0**
 - per il processo padre è un valore **positivo** che rappresenta il **PID** del processo figlio
 - è un valore **negativo** in caso di **errore** (la creazione non è andata a buon fine)

Effetti della fork()

- ❑ **Allocazione** di una **nuova process structure** nella process table associata al processo figlio e sua **inizializzazione**
- ❑ **Allocazione** di una **nuova user structure** nella quale viene **copiata la user structure del padre**
- ❑ **Allocazione** dei **segmenti di dati e stack** del figlio nei quali vengono **copiati dati e stack del padre**
- ❑ Aggiornamento del riferimento **text** al codice eseguito (**condiviso col padre**): incremento del contatore dei processi, ...

Effetti della fork()



Relazione padre-figlio in UNIX

Dopo una `fork()`:

- **concorrenza**
 - padre e figlio procedono contemporaneamente
- **lo spazio degli indirizzi è duplicato (ma il codice è condiviso)**
 - ogni **variabile del figlio è inizializzata con il valore assegnatole dal padre** prima della `fork()`
- **la user structure è duplicata:**
 - le **risorse allocate al padre** (ad esempio, i file aperti) prima della generazione sono **condivise con i figli**
 - le informazioni per **la gestione dei segnali** sono le stesse per padre e figlio (associazioni segnali-handler)
 - il figlio nasce con lo **stesso program counter del padre**: la **prima istruzione** eseguita dal figlio è **quella che segue immediatamente fork()**

Esecuzioni differenziate del padre e del figlio

```
...
if (fork() == 0) {
    ... /* codice eseguito dal figlio */
    ...
} else {
    ... /* codice eseguito dal padre */
    ...
}
```

Dopo la `fork()`:

- **il figlio** inizia la sua esecuzione **dall'istruzione immediatamente successiva alla `fork`** che l'ha creato.
- **il padre** continua la sua esecuzione **concorrentemente al figlio**.

fork(): esempio

```
#include <stdio.h>
main()
{ int pid;
  pid=fork();
  if (pid==0)
  { /* codice figlio */
    printf("Sono il figlio ! (pid: %d)\n", getpid());
  }
  else if (pid>0)
  { /* codice padre */
    printf("Sono il padre: pid di mio figlio: %d\n", pid);
  }
  else printf("Creazione fallita!");
}
```

NB: system call **getpid()** ritorna il pid del processo che la chiama

Terminazione di processi

Un processo può terminare:

- **involontariamente**

- tentativi di **azioni illegali**
- interruzione mediante **segnale**
 - 👉 salvataggio dell'immagine nel file **core**

- **volontariamente**

- esecuzione dell'ultima istruzione
- chiamata alla funzione **exit()**

exit()

```
void exit(int status);
```

- la funzione **exit()** prevede un parametro (**status**) mediante il quale il processo che termina può comunicare al padre **informazioni sul suo stato di terminazione** (ad esempio esito dell'esecuzione)
- è **sempre una chiamata senza ritorno**

exit()

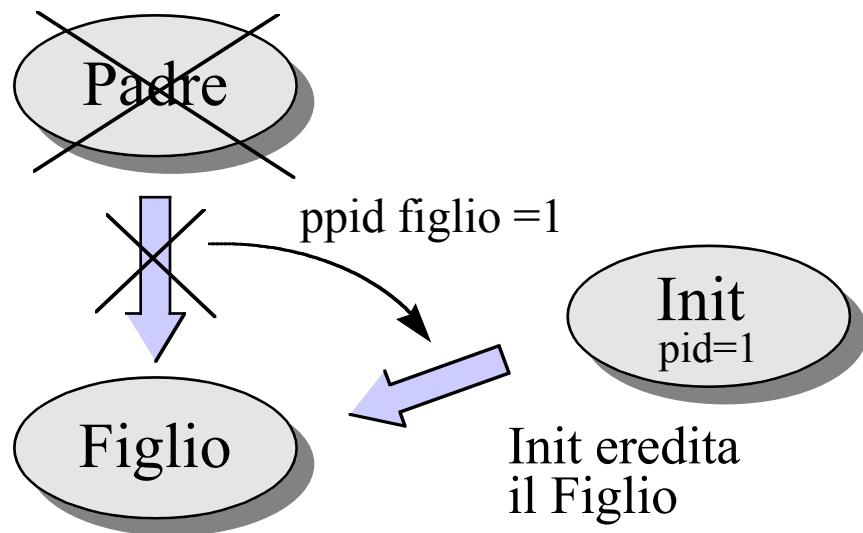
Effetti di una exit():

- *chiusura dei file aperti non condivisi*
- *terminazione del processo:*
 - » se il processo che termina ha **figli in esecuzione**, il processo **init adotta i figli dopo la terminazione del padre** (nella process structure di ogni figlio al pid del processo padre viene assegnato il valore 1)
 - » se il processo **termina prima che il padre ne rilevi lo stato di terminazione** con la system call **wait()**, il processo passa nello stato **zombie**

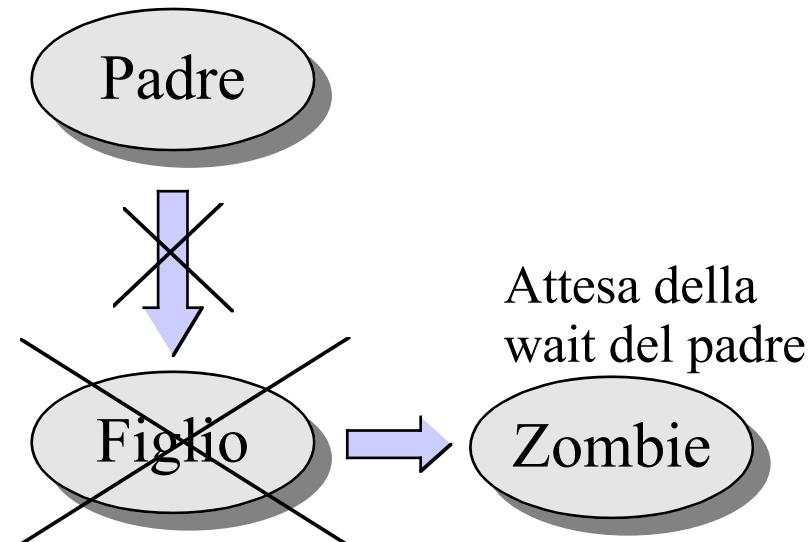
NB: Quando termina un processo adottato dal processo **init**, **init** rileva automaticamente il suo stato di terminazione -> i processi figli di **init** non permangono nello stato di zombie

Parentela processi e terminazione

Terminazione del padre



Terminazione del figlio: processi zombie



wait()

Lo stato di terminazione può essere rilevato dal processo padre, mediante la system call **wait()**

```
int wait(int *status);
```

- ❑ parametro **status** è **l'indirizzo** della variabile in cui viene memorizzato lo **stato di terminazione del figlio**
- ❑ risultato prodotto dalla **wait()** è **pid del processo terminato**, oppure un codice di errore (<0)

wait()

Effetti della system call wait(&status):

- ❑ processo che la chiama può avere figli in esecuzione:
 - ✓ se tutti i figli non sono ancora terminati, il processo si **sospende in attesa della terminazione del primo** di essi
 - ✓ se almeno un figlio F è già terminato ed il suo stato non è stato ancora rilevato (cioè F è in stato **zombie**), **wait() ritorna immediatamente con il suo stato di terminazione** (nella variabile **status**)
 - ✓ se non esiste neanche un figlio, **wait() NON è sospensiva** e ritorna un codice di errore (valore ritornato < 0)

wait()

Rilevazione dello stato: in caso di terminazione di un figlio, la variabile status raccoglie stato di terminazione. Nell'ipotesi che lo stato sia un intero a 16 bit:

- se il byte meno significativo di status è zero, il più significativo rappresenta lo **stato di terminazione** (**terminazione volontaria**, ad esempio con **exit**)
- in caso contrario, il byte meno significativo di status descrive il **segnale che ha terminato il figlio** (**terminazione involontaria**)

wait() & exit(): esempio

```
main()
{
    int pid, status;
    pid=fork();
    if (pid==0)
        {printf("figlio");
        exit(0);
    }
    else{ pid = wait(&status);
        printf("terminato processo figlio n.%d", pid);
        if ((char)status==0)
            printf("term. volontaria con stato %d", status>>8);
        else printf("terminazione involontaria per segnale
                    %d\n", (char)status);
    }
}
```

wait()

Rilevazione dello stato: è necessario conoscere la rappresentazione di **status**

- ❑ lo standard POSIX.1 prevede delle macro (definite nell'header file **<sys/wait.h>**) per l'analisi dello stato di terminazione. In particolare
 - ✓ **WIFEXITED(status)**: restituisce **vero** se il processo figlio è terminato volontariamente. In questo caso la macro **WEXITSTATUS(status)** restituisce lo stato di terminazione
 - ✓ **WIFSIGNALED(status)**: restituisce **vero** se il processo figlio è **terminato involontariamente**. In questo caso la macro **WTERMSIG(status)** restituisce il numero del segnale che ha causato la terminazione

wait() & exit(): esempio

```
#include <sys/wait.h>
main()
{
    int pid, status;
    pid=fork();
    if (pid==0)
        {printf("sono il figlio\n");
         exit(0);
    }
    else { pid=wait(&status);
            if (WIFEXITED(status))
                printf("Terminazione volontaria di %d con
                       stato %d\n", pid, WEXITSTATUS(status));
            else if (WIFSIGNALED(status))
                printf("terminazione involontaria per segnale
                       %d\n", WTERMSIG(status));  } }
```

Esempio con più figli

```
#include <sys/wait.h>
#define N 100
int main()
{ int pid[N], status, i, k;
  for (i=0; i<N; i++)
  { pid[i]=fork();
    if (pid[i]==0)
    { printf("figlio: il mio pid è: %d", getpid());
      exit(0);
    }
  }
```

```
/* continua (codice padre) . . */

for (i=0; i<N; i++) /* attesa di tutti i figli */
{ k=wait(&status);
    if (WIFEXITED(status))
        printf("Term. volontaria di %d con
                stato %d\n", k,
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("term. Involontaria di %d per
                segnale %d\n", k, WTERMSIG(status));
}
```

System call exec()

Mediante **fork()** i processi **padre e figlio condividono il codice e lavorano su aree dati duplicate**. In UNIX è possibile **differenziare il codice dei due processi** mediante una system call della famiglia **exec**:

execl(), execle(), execlp(), execv(), execve(), execvp()...

Effetto principale di system call famiglia exec:

- ✓ vengono **sostituiti codice ed eventuali argomenti di invocazione** del processo che chiama la system call, **con codice e argomenti di un programma specificato come parametro** della system call

NO generazione di nuovi processi

exec()

```
int exec(char *pathname, char *arg0, ..  
        char *argN, (char*)0);
```

- ✓ **pathname** è il nome (assoluto o relativo) dell'eseguibile da caricare
- ✓ **arg0** è il nome del programma (argv[0])
- ✓ **arg1, ..., argN** sono gli argomenti da passare al programma
- ✓ **(char *)0** è il puntatore nullo che termina la lista

Utilizzo system call exec()

(differenziare il comportamento del padre da quello del figlio)

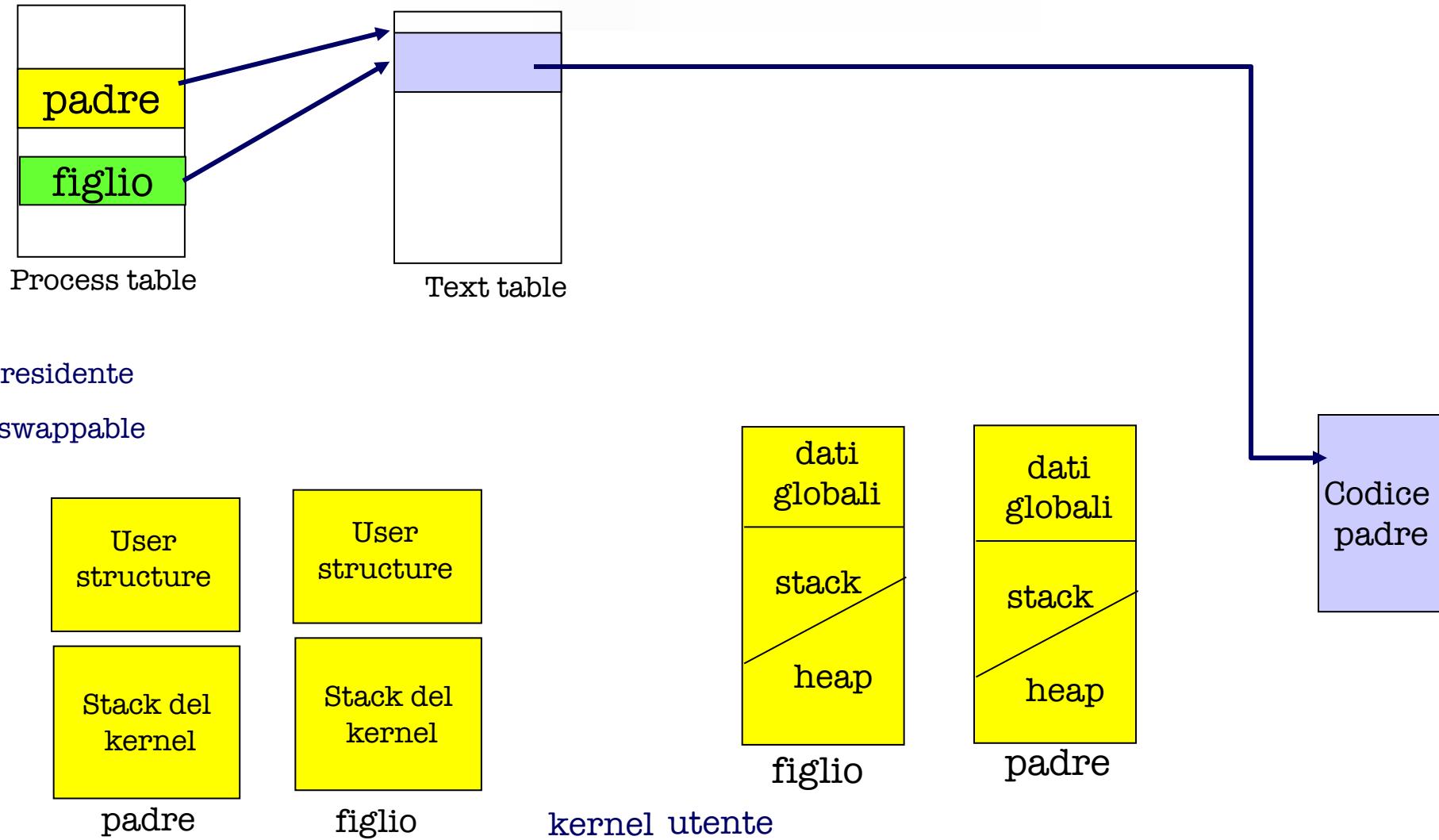
```
pid = fork();
if (pid == 0){ /* figlio */
    printf("Figlio: esecuzione di ls\n");
    execl("/bin/ls", "ls", "-l", (char *)0);
    printf("Errore in execl\n");
    exit(1); }
if (pid > 0){ /* padre */
    ...
    printf("Padre ....\n");
    exit(0); }
if (pid < 0){ /* fork fallita */
    print("Errore in fork\n");
    exit(1); }
```

Il figlio passa a **eseguire** un altro programma: si caricano il nuovo codice e gli argomenti per il nuovo programma

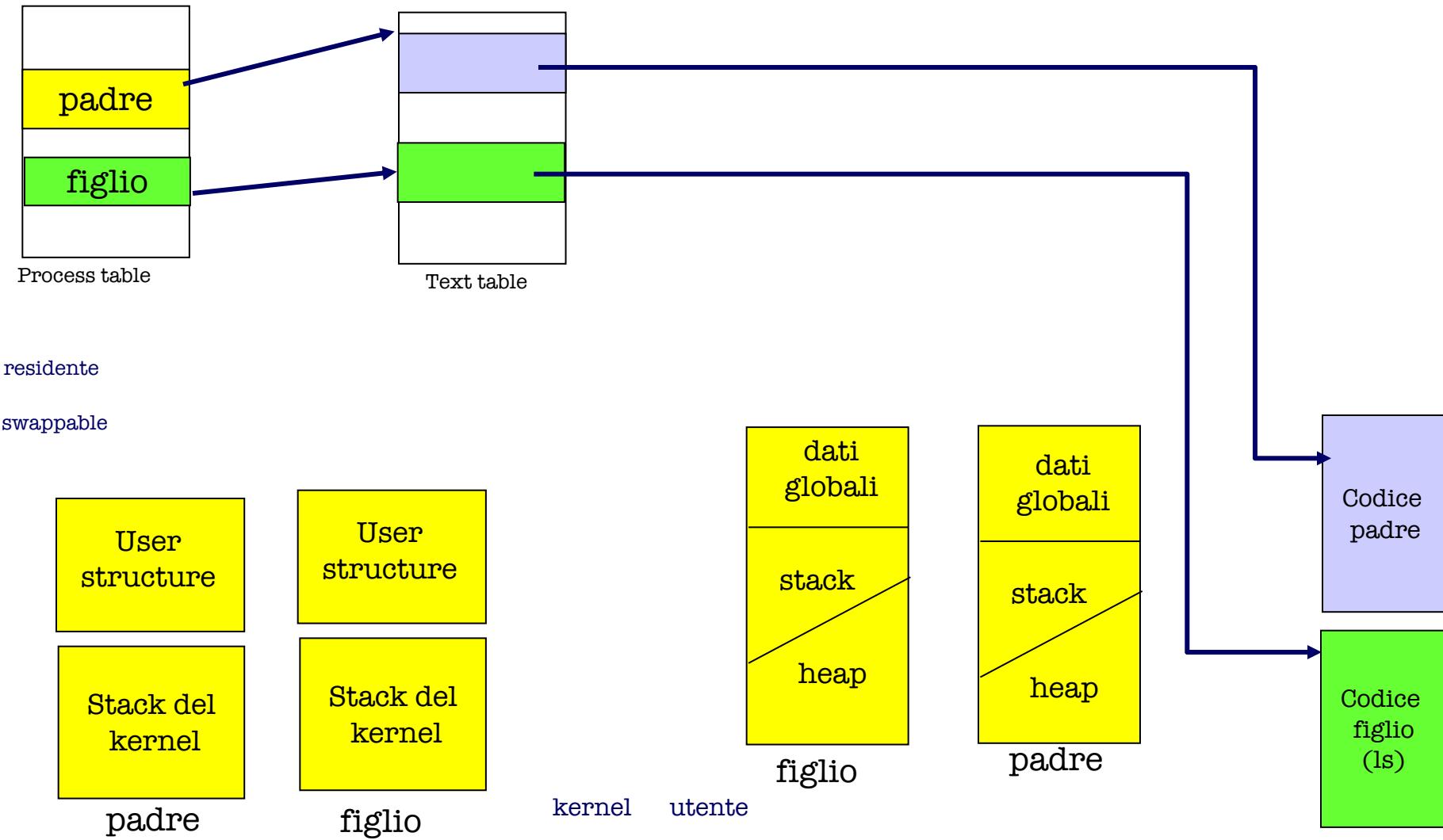
Si noti che exec è operazione senza ritorno

Esempio: effetti della exec()

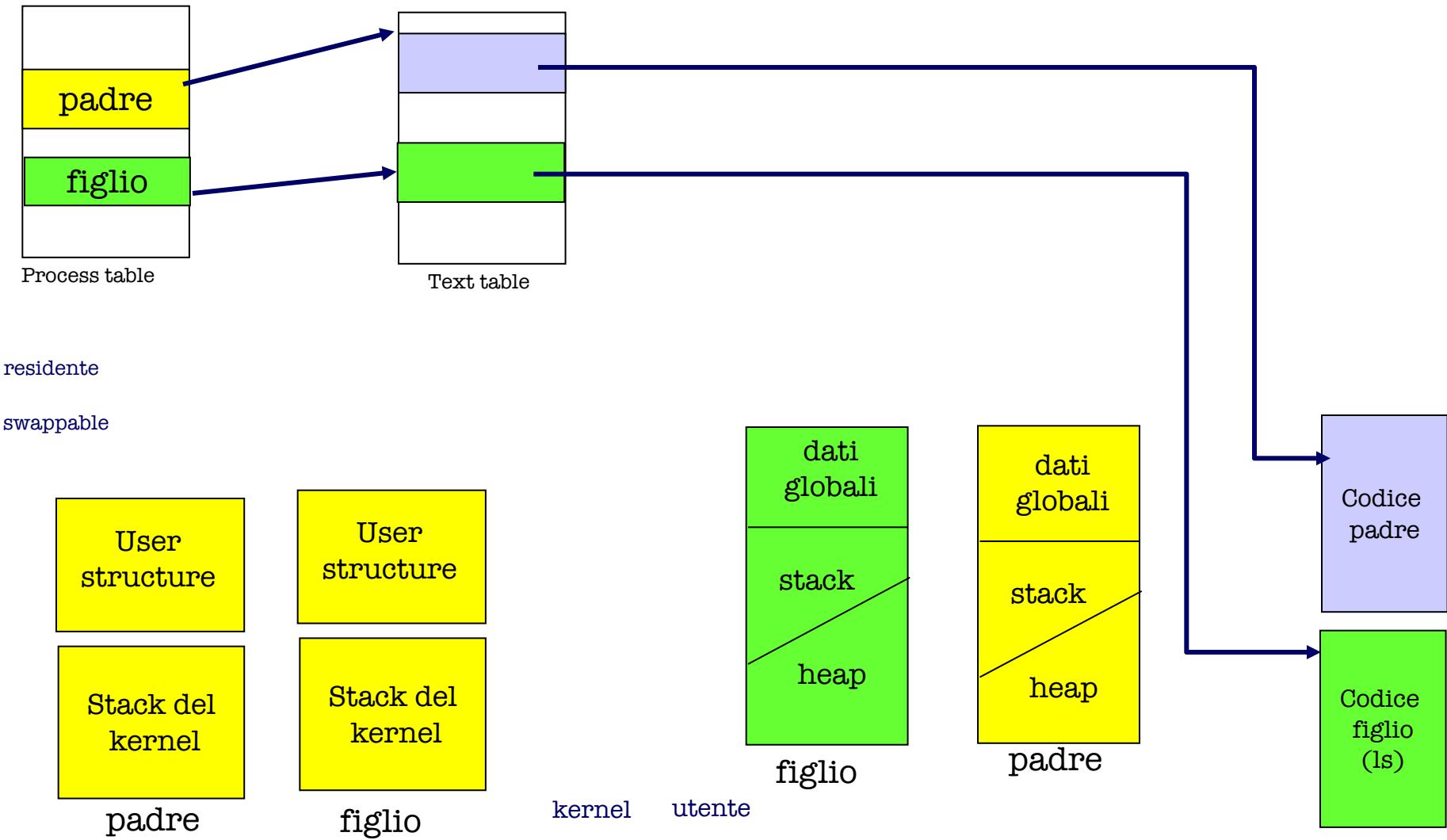
sull'immagine



Esempio: effetti della exec()



Esempio: effetti della exec()



Effetti dell'exec()

Il processo dopo exec()

- ❑ mantiene la ***stessa process structure*** (salvo le informazioni relative al codice):
 - » stesso pid
 - » stesso pid del padre
 - » ...
- ❑ ha ***codice, dati globali, stack e heap*** nuovi
- ❑ riferisce un ***nuovo text***
- ❑ mantiene ***user structure (a parte PC e informazioni legate al codice) e stack del kernel:***
 - » mantiene le stesse risorse (es: file aperti)
 - » mantiene lo stesso *environment* (a meno che non sia **`execle`** o **`execve`**)

System call exec()

Varianti di exec, a seconda del suffisso

- l** gli argomenti da passare al programma da caricare vengono specificati mediante una ***LISTA di parametri (terminata da NULL)*** – es. **execl()**
- p** il nome del file eseguibile specificato come argomento della system call viene ricercato nel ***PATH contenuto nell'ambiente*** del processo – es. **execvp()**
- v** gli argomenti da passare al programma da caricare vengono specificati mediante un ***VETTORE di parametri*** – es. **execv()**
- e** la system call riceve anche un ***vettore (envp[]) che rimpiazza l'environment*** (path, direttorio corrente, ...) del processo chiamante – es. **execle()**

Esempio: execve()

```
int execve(char *pathname, char *argv[], char * env[]);
```

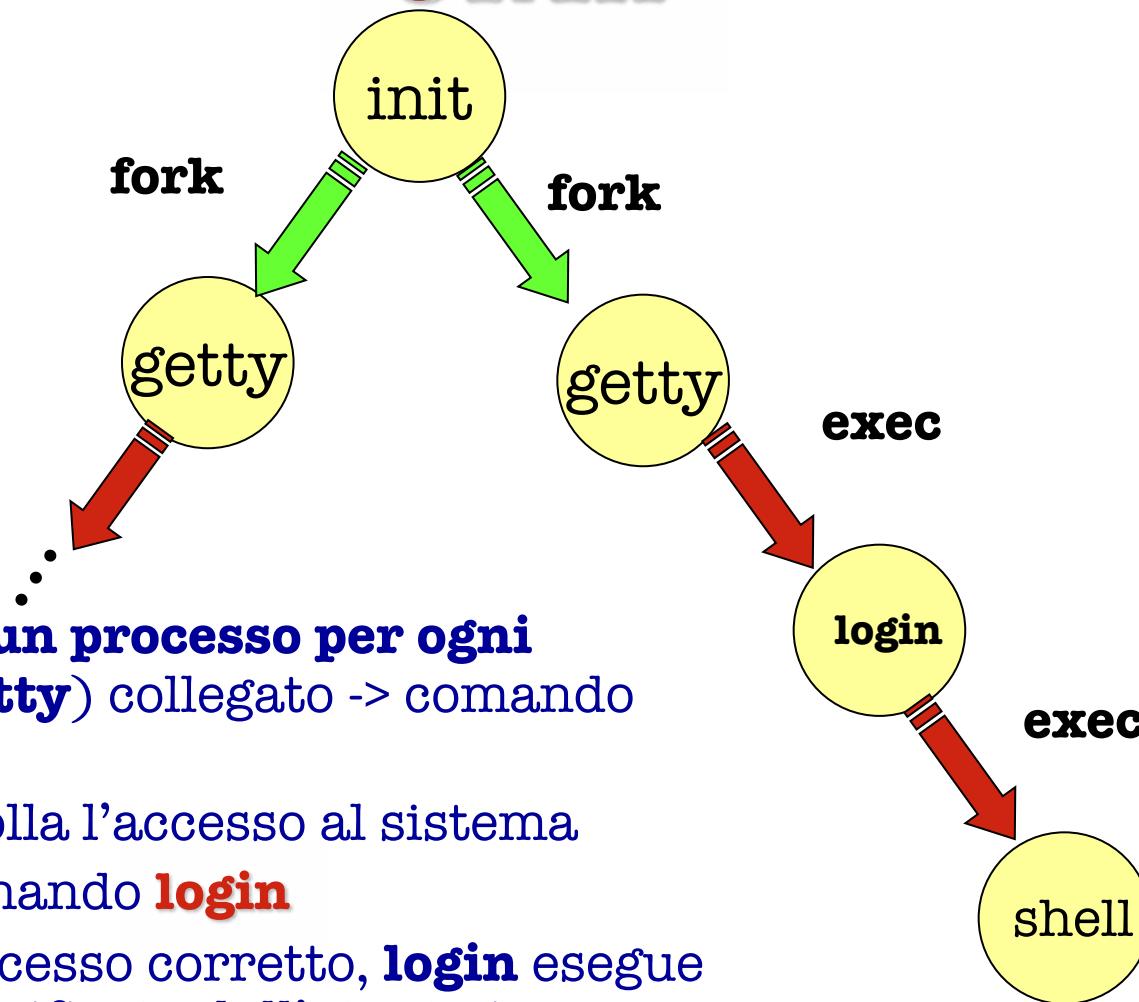
- ✓ **pathname** è il **nome** (assoluto o relativo) **dell'eseguibile** da caricare
- ✓ **argv** è il **vettore degli argomenti** del programma da eseguire
- ✓ **env** è il **vettore delle variabili di ambiente** da sostituire all'ambiente del processo (contiene stringhe del tipo “VARIABILE=valore”)

Esempio: execve()

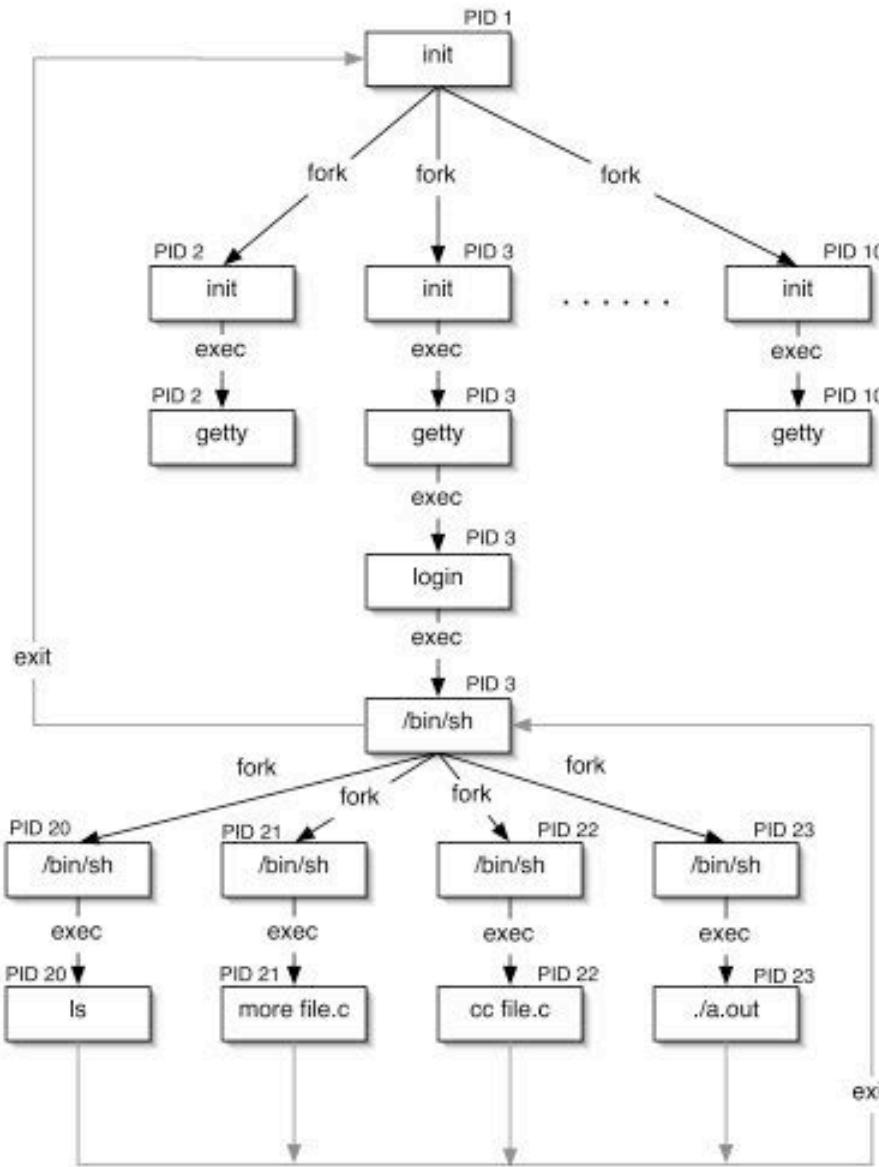
```
char *env[]={"USER=paolo", "PATH=/home/paolo/d1", (char *)0} ;  
char *argv[]={"ls", "-l", "pippo", (char *)0} ;  
  
int main()  
{ int pid, status;  
    pid=fork();  
    if (pid==0)  
    {   execve("/bin/ls", argv, env);  
        printf("exec fallita!\n");  
        exit(1);  
    }  
    else if (pid >0)  
    {   pid=wait(&status); /* gestione dello stato.. */  
    }  
    else printf("fork fallita!\n");  
}
```

Inizializzazione dei processi

UNIX

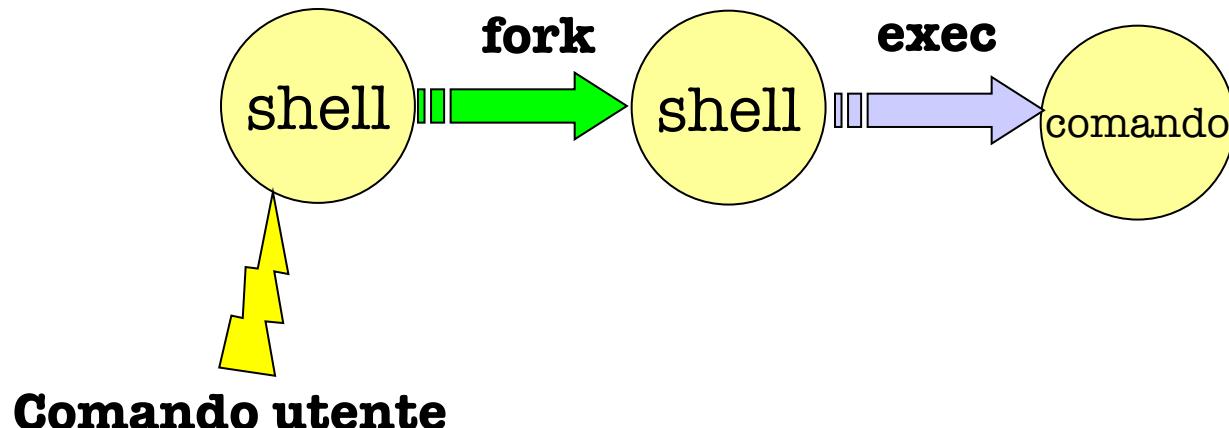


Tipico albero di generazione di processi



Interazione con l'utente tramite shell

- Ogni utente può interagire con lo **shell** mediante la **specifica di comandi**
- Ogni **comando** è presente nel file system come **file eseguibile** (direttorio **/bin**)
- Per ogni comando, **shell genera un processo figlio** dedicato all'esecuzione del comando:



Relazione shell padre-shell figlio

Per ogni comando, shell genera un figlio; possibilità di **due diversi comportamenti**:

- il padre si pone in attesa della terminazione del figlio (esecuzione in ***foreground***); es:

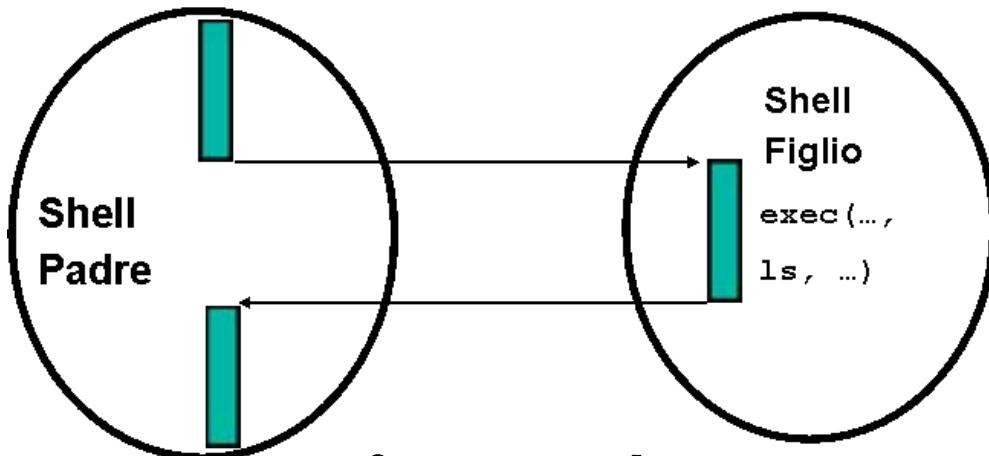
ls -l pippo

- il padre continua l'esecuzione concorrentemente con il figlio (esecuzione in ***background***):

ls -l pippo &

foreground vs background

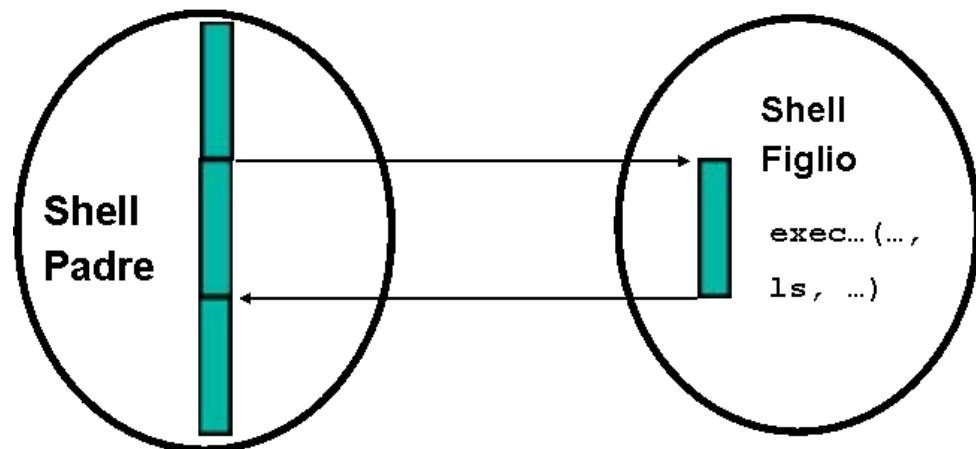
\$ ls



foreground

\$ ls&

background



ESERCIZIO (esecuzione di comandi “in foreground”)

```
#include <stdio.h>
int main (argc, argv) {
    int stato, atteso, pid; char st[80];
    for (;;) {
        if ((pid = fork()) < 0) {perror("fork");
            exit(1);}
        if (pid == 0) { /* FIGLIO: esegue i comandi */
            printf("inserire il comando da eseguire:\n");
            scanf ("%s", st);
            execp(st, st, (char *)0);
            perror("errore");/* stampa messaggio errore*/
            exit (0);
        } else { /* PADRE */
            atteso=wait (&stato);
            /*attesa figlio: sincronizzazione */
            printf ("eseguire altro comando? (si/no) \n");
            scanf ("%s", st);
            if (strcmp(st, "si") == 0) exit(0); } } }
```

Gestione degli errori: perror()

Convenzione:

- in caso di fallimento, ogni **system call ritorna un valore negativo** (tipicamente, -1)
- in aggiunta, UNIX prevede la variabile globale di sistema **errno**, alla quale il kernel assegna il codice di errore generato dall'ultima system call eseguita. Per interpretarne il valore è possibile usare la funzione **perror()**:
 - **perror("stringa")** stampa "stringa" seguita dalla descrizione del codice di errore contenuto in **errno**
 - la corrispondenza tra codici e descrizioni è contenuta in

<sys/errno.h>

perror()

```
int main()
{int pid, status;
pid=fork();
if (pid==0)
{execl(“/home/paolo/prova”, “prova”, (char *)0);
perror(“exec fallita a causa dell’errore:”);
exit(1);
}
```

...

Esempio di output:

```
exec() fallita a causa dell’errore: No such file or directory
```

Esercizio sulle fifo

Esercizio

Si vuole realizzare una applicazione Unix basata sullo schema «cliente-servitore».

L'applicazione deve essere costituita da due processi (cliente e servitore, non necessariamente parenti):

- ❑ il **cliente** è un programma che prevede la sintassi di invocazione:

```
./cliente com fileout
```

Il cliente interagisce con il servitore per richiedere l'esecuzione di un servizio (l'esecuzione del comando **com**) con ridirezione sul file **fileout**.

- ❑ il **servitore** è un programma (che non prevede argomenti), che, una volta ricevuti dal cliente **com** e **fileout**, provvede all'esecuzione di **com** con ridirezione su **fileout**.

Impostazione

- Processi non appartenenti alla stessa gerarchia devono comunicare:
 - 👉 la comunicazione non può avvenire mediante pipe
→ utilizziamo **fifo**
- Il servitore deve essere attivo e pronto ad accettare la richiesta dal cliente-
- Il cliente prende l'iniziativa nella comunicazione.

Soluzione dell'esercizio: struttura del servitore

```
#include <fcntl.h>
#include <signal.h>
#define msgdim 16
int fdin;

main()
{
    int pid, stato;char com[msgdim], fileout[msgdim];
    if((fdin=mkfifo("server.in", 0777))<0)
        perror("server- mkfifo in"); /*la fifo esiste già*/
    if ((fdin=open("server.in", O_RDONLY))<0)
    {
        perror("server_open in:");
        exit(-1);
    }
```

```
read(fdin,com,msgdim); /*lettura comando */
read(fdin,fileout,msgdim);/*lettura nome file */
close(fdin);
pid=fork();
if (!pid)
{ close(1);
  fdin=creat(fileout, 0777); /* rid. output */
  execlp(com, com, (char *)0);
  perror("execlp");
  exit(-1);}
else
{ wait(&stato);
  printf("stato figlio %d\n", stato>>8);
  exit(0);
}
}
```

Soluzione dell'esercizio: struttura del cliente

```
#include <fcntl.h>

#define msgdim 16

main(int argc, char **argv)
{int fd;char com[msgdim], fileout[msgdim];
 if (argc!=3){printf("errore sintattico");
 exit(-1);
 sprintf(com,"%s",argv[1]);
 sprintf(fileout,"%s", argv[2]);
 if ((fd=open("server.in", O_WRONLY))<0)
 { perror("client-open fifo in: ");
 exit(-1);
 }
```

```
write(fd, com, msgdim); /* send com */
write(fd,fileout,msgdim); /* send fileout */
close(fd);
unlink("server.in");
}
```

Spunti per estensioni e modifiche

- Generalizzare server:
 - capacità di gestire più richieste
- Estendere lo schema di comunicazione:
 - invece di ridirigere l'output del comando su file, restituire l'output al cliente mediante ridirezione su fifo.